



Python带我起飞

入门、进阶、商业实战

李金洪 编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书针对 Python 3.5 以上版本,采用“理论+实践”的形式编写,通过大量的实例(共 42 个),全面而深入地讲解“Python 基础语法”和“Python 项目应用”两方面内容。书中的实例具有很强的实用性,如对医疗影像数据进行分析、制作爬虫获取股票信息、自动化实例、从一组看似混乱的数据中找出规律、制作人脸识别系统等。

书中的每章都配有同步的教学视频。视频和图书具有相同的结构,能帮助读者快速而全面地了解本章的内容。本书还免费提供了所有案例的源代码及素材样本。这些代码和素材样本不仅方便了读者学习,而且也能为以后的工作提供便利。

全书共分为 4 篇:第 1 篇,包括了解 Python、配置机器及搭建开发环境、语言规则;第 2 篇,介绍了 Python 语言的基础操作,包括变量与操作、控制流、函数操作、错误与异常、文件操作;第 3 篇,介绍了更高级的 Python 语法知识及应用,包括面向对象编程、系统调度编程;第 4 篇,是前面知识的综合应用,包括爬虫实战、自动化实战、机器学习实战、人工智能实战。

本书结构清晰、案例丰富、通俗易懂、实用性强。特别适合 Python 语言的初学和进阶读者,作为自学教程阅读。另外,本书也适合社会培训学校作为培训教材使用,还适合大中专院校的相关专业作为教学参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Python 带我起飞:入门、进阶、商业实战 / 李金洪编著. —北京:电子工业出版社, 2018.6

ISBN 978-7-121-34322-3

I. ①P… II. ①李… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 111614 号

策划编辑:吴宏伟

责任编辑:牛 勇

印 刷:

装 订:

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×980 1/16 印张:21.75 字数:528 千字 彩插:2

版 次:2018 年 6 月第 1 版

印 次:2018 年 6 月第 1 次印刷

定 价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

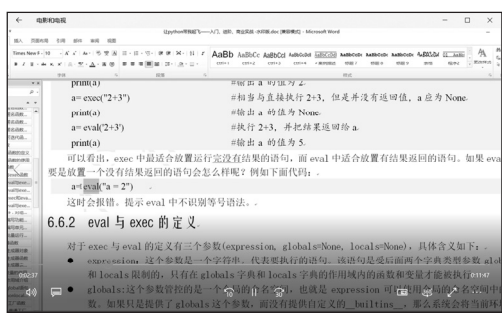
本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

配套学习资源

本书提供了大量的超值学习资料（下载方法见封底），下面分别介绍。

1. 同步配套教学视频

作者按照图书的内容和结构，录制了同步对应的教学视频，如图 1 所示。既有上课式讲解，又有具体的代码实操。



电量和电压

```
def func(x):  
    # 返回 x 的平方  
    a = exec("2+3")  
    print(a)  
    # 点击 a 时值为 None  
    a = eval("2+3")  
    print(a)  
    # 执行 2+3，并返回结果 5
```

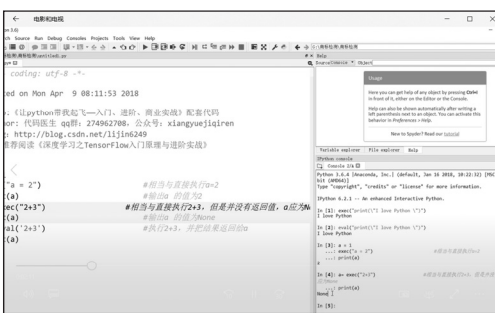
可以看出，exec 中最适合放置运行没有结果的语句，而 eval 中适合放置有结果返回的语句。如果 eval 要是放置一个没有结果返回的语句会怎么样呢？例如下面代码：..

```
a = eval("a = 2")  
这时会报错，提示 eval 中不识别等号语法。..
```

6.6.2 eval 与 exec 的定义。

对于 exec 与 eval 的定义有三个参数(expression, globals=None, locals=None)，具体含义如下：

- expression: 这个参数是一个字符串，代表要执行的语句。该语句是受后面两个字典类型参数 a 和 locals 限制的，只有在 globals 字典和 locals 字典的作用域内的函数和变量才能被执行。
- globals: 这个参数指的是一个字典的命名空间，也就是 expression 可以引用字典中的变量。
- locals: 如果提供了 globals 这个参数，那么对 locals 的定义，locals 是必须提供的。



电量和电压

```
def func(x):  
    # 返回 x 的平方  
    a = exec("2+3")  
    print(a)  
    # 点击 a 时值为 None  
    a = eval("2+3")  
    print(a)  
    # 执行 2+3，并返回结果 5
```

可以看出，exec 中最适合放置运行没有结果的语句，而 eval 中适合放置有结果返回的语句。如果 eval 要是放置一个没有结果返回的语句会怎么样呢？例如下面代码：..

```
a = eval("a = 2")  
这时会报错，提示 eval 中不识别等号语法。..
```

6.6.2 eval 与 exec 的定义。

对于 exec 与 eval 的定义有三个参数(expression, globals=None, locals=None)，具体含义如下：

- expression: 这个参数是一个字符串，代表要执行的语句。该语句是受后面两个字典类型参数 a 和 locals 限制的，只有在 globals 字典和 locals 字典的作用域内的函数和变量才能被执行。
- globals: 这个参数指的是一个字典的命名空间，也就是 expression 可以引用字典中的变量。
- locals: 如果提供了 globals 这个参数，那么对 locals 的定义，locals 是必须提供的。

对着书稿,像老师上课一样讲解

具体代码实操

| | | | | | | | | | |
|---|---|---|--|---|--|--|---|--|---|
| python视频: 2 开发环境 -2.1-2.2版本及 安装.mp4 | python视频: 2 开发环境-2.3熟 悉开发环境. mp4 | python视频: 2 开发环境-2.4熟 悉1展示程序. mp4 | python视频: 3 语法规则 -3.1-3.2基础规 则.mp4 | python视频: 3 语法规则 -3.3-3.4文件结 构及模块.mp4 | python视频: 3 语法规则-3.5模 块导入方式. mp4 | python视频: 3 语法规则-3.6实 例2导入模块. mp4 | python视频: 4 变量-4.1什么是 变量.mp4 | python视频: 4 变量-4.2了解交 量的规则.mp4 | python视频: 4 变量 -4.3numbers类型. mp4 |
| python视频: 4 变量-4.4string 类型.mp4 | python视频: 4 变量-4.5列表类 型-实例1列表. mp4 | python视频: 4 变量-4.5列表类 型-实例1列表. mp4 | python视频: 4 变量-4.6元组类 型.mp4 | python视频: 4 变量-4.7集合类 型.mp4 | python视频: 4 变量-4.8字典类 型.mp4 | python视频: 4 变量-4.9深浅拷 贝.mp4 | python视频: 5 控制流-5.1if语 句.mp4 | python视频: 5 控制流-5.2while语 句.mp4 | python视频: 5 控制流-5.3for语 句.mp4 |
| python视频: 5 控制流-5.4-5.5 循环控制语句. mp4 | python视频: 5 控制流-5.6-5.7 列表表达式. mp4 | python视频: 6 函数基本操作 -6.1函数的基本 概念.mp4 | python视频: 6 函数基本操作 -6.2定义及调用 函数.mp4 | python视频: 6 函数基本操作 -6.3匿名函数与 可迭代函数.m p4 | python视频: 6 函数基本操作 -6.4函数参数 传递.mp4 | python视频: 6 函数基本操作 -6.5eval与exec 19.mp4 | python视频: 6 函数基本操作 -6.6实例 19.mp4 | python视频: 6 函数基本操作 -6.8函数类型 6.9函数.m p4 | python视频: 6 函数基本操作 -6.10工厂函数. mp4 |
| python视频: 7 错误与异常-7.1 错误分类7.2异 常的基本语法.. mp4 | python视频: 7 错误与异常-7.3 常见异常7.4创 建异常.mp4 | python视频: 7 错误与异常 -7.5finally7.6 异常7.7实例.m p4 | python视频: 8 文件操作-8.1基 本操作8.2文件 对象方法.m p4 | python视频: 8 文件操作-8.3实 例8.4with语句 10.mp4 | python视频: 8 文件操作-8.5二 进制转化8.6序 列化.mp4 | python视频: 9 类-9.1类的基本 概念9.2实例化 对象.mp4 | python视频: 9 类-9.2实例化对 象.mp4 | python视频: 9 类-9.3类属性 9.4类-9.4类 mp4 | python视频: 9 类-9.4类 mp4 |
| python视频: 9 类-9.5类属性 9.6类方法.m p4 | python视频: 9 类-9.7类属性 9.8类方法.m p4 | python视频: 9 类-9.8类属性 9.9类方法.m p4 | python视频: 9 类-9.9类属性 9.10类方法.m p4 | python视频: 9 类-9.10类属性 9.11类方法.m p4 | python视频: 9 类-9.11类属性 9.12类方法.m p4 | python视频: 9 类-9.12类属性 9.13类方法.m p4 | python视频: 9 类-9.13类属性 9.14类方法.m p4 | python视频: 9 类-9.14类属性 9.15类方法.m p4 | python视频: 9 类-9.15类属性 9.16类方法.m p4 |

第 2~10 章配套视频
共计 47 段, 总时长
429 分钟

图 1 本书教学视频

2. 书中案例的源文件

本书提供了书中涉及的所有案例的源文件，如图 2 所示。读者可以一边阅读本书，一边参照源文件动手练习。这样不仅提高了学习的效率，而且可以对书中的内容有更直观的认识，从而逐渐培养自己的编码能力。



图 2 本书案例源文件

3. 书中案例用到的素材和样本

书中案例用到的素材和样本也都全部提供了。读者可以采用这些素材和样本，完全再现书中的案例效果，如图 3 所示。



图 3 本书案例用到的素材和样本

4. 加入学习 QQ 群，与千人成为同学。即时交流，共享资源

QQ 群：274962708。图 4 所示为群内交流情况。

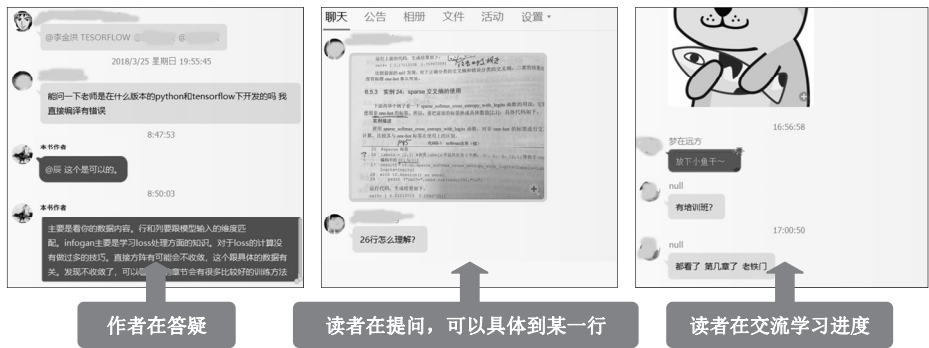


图 4 群内交流情况

书 评

在日常的工作中，处理报表、简单的网络测试这类相对零散的小任务，使用 Python 都可以很快实现。《Python 带我起飞——入门、进阶、商业实战》，是一本非常实用的技术书籍。在我们公司内部，本书内容得到了研发人员的一致好评。强烈推荐！

北京派网软件有限公司 CEO 孙朝晖

终于盼到李金洪的这本书了。该书从浅到深地讲解了 Python 的“神奇功能”，知识点很体系化，内容很丰富。书中的知识在信息安全领域中也非常适用。与此同时，本书也非常适合软件工程、网络工程、人工智能、网络信息安全攻防等多领域有需求的相关群体阅读。很值得推荐给新手用作入门学习。

孔韬循（网名：K0r4dji）

北京丁牛科技有限公司首席安全官

国内著名信息安全团队——破晓安全团队创始人
人民邮电出版社-异步社区-信息安全领域图书专家顾问

Python 已经成为人工智能（AI）的第一语言。《Python 带我起飞——入门、进阶、商业实战》这本书可以让读者的 Python 基础变得更牢固，为后续的 AI 开发之路扫平障碍。并且，后几章还有关于数据处理的实例，是加固 Python 基础的不二之选。强烈推荐！

佳格天地公司首席科学家，马里兰大学博士 宋宽

本书通俗易懂，内容全面，非常适合学习。强烈推荐！

盛大游戏传奇工作室研发总监 宫晓峰

VI | Python 带我起飞：入门、进阶、商业实战

学习编程语言最怕的是：书看了，知识懂了，一上手却懵了。作者将多年 Python 开发的宝贵经验凝聚书中，没有采用说教式的知识灌输，而是在每个知识点后融入了日常工作中经常会用到的实战案例。读者不仅仅学习到了知识点，同时还掌握了知识点可以用在哪、怎样用。相信通过阅读本书，不论是初学者还是老手，都会收获满满。

赛宁网安副总经理，Vulhub（原 SCAP 社区）创始人 王珩

使用 Python 语言开发程序，可以大大节省时间。《Python 带我起飞——入门、进阶、商业实战》一书可以大大节省读者学习 Python 的时间。即便是对已经掌握 Python 的人，本书也可作为语法速查工具书。强烈推荐。

腾讯视频智能应用中心副总监 杨广煜

大数据时代，使用 Python 开发程序更加高效、便捷。只有更细致而全面地掌握 Python 语言，才能在工作中表现出惊人的效率，更好地表现出自我价值。感谢《Python 带我起飞——入门、进阶、商业实战》这本书，使我们认识了不一样的 Python。

阿里巴巴广告算法高级专家 易慧民

Python 作为脚本语言，在工作中必不可少，它可以大大的提升工作效率，《Python 带我起飞——入门、进阶、商业实战》这本书，内容非常全面，案例丰富，条理清晰，在工作之中，伴随左右，时常查阅，帮助很大。强烈推荐。

百度前高级研发工程师，现任深圳大旺网络科技有限公司总经理 邓云鹏

在这个 DT 时代，面对海量的数据，如何才能快速、有效地解决问题？Python 是开发语言中的第一选择。《Python 带我起飞——入门、进阶、商业实战》这本书层层递进的解析了 Python 语言，同时还告诉你各种实际问题场景的解决方法，是一本不可多得之书，让你真正认识 Python 的力量，让工作更高效。

滴滴项目管理专家 李斌

《Python 带我起飞——入门、进阶、商业实战》主要是对 Python 的基础语法进行讲解，覆盖全面，细致深入，是一本很容易上手的好书。强烈推荐。

“什么值得买”网大数据高级开发工程师 陆建强

在自动化项目中，Python 是必备语言。《Python 带我起飞——入门、进阶、商业实战》这本书，作为基础的语法工具书对我们帮助很大。它可以使读者更全面系统地了解语法规则，写出更高效、简洁的代码，大大缩短了开发周期。强烈推荐。

华为项目经理 张欲涛

《Python 带我起飞——入门、进阶、商业实战》一书，拉近了 Python 与初学者的距离。书中的案例非常实用，具有很大的参考价值。强烈推荐。

魅族前高级开发工程师 缪斯

《Python 带我起飞——入门、进阶、商业实战》条理清晰、浅显易懂，是初学者快速上手 Python 语言，实现“从入门到精通”的优质书籍。

滴滴平台技术部高级开发工程师 曹春晖

Python 语言对于编程来讲，就相当于财务软件对于办公自动化的位置。《Python 带我起飞——入门、进阶、商业实战》这本书就相当于财务软件的操作手册，可以使读者快速掌握并且应用，大大地提高工作效率。强烈推荐。

用友前高级开发工程师 高翔

前 言

目前，人工智能技术在现代社会中的地位日趋重要，尤其在自动化和数据驱动的诸多领域，如图像识别、机器人学、搜索引擎、自动驾驶技术都有不俗的表现。Python 语言借助 AI 和数据科学，攀爬到了编程语言生态链的顶级位置，成为应用最火，风头正劲的开发语言。

除了人工智能方向，Python 语言还可以更高效地应用在 Web 应用开发、图形界面开发、系统网络运维、网络编程、科学与数字计算、3D 游戏开发等诸多领域。

Python 语言简洁优美，开发效率极高，得到了越来越多公司的青睐。

“人生苦短，我用 Python” 这样的佳话也在业内广为流传。

不仅仅是业内企业，包括国家的教育机构也对 Python 语言高度重视：

- 教育部考试中心在计算机二级考试中加入了“Python 语言程序设计”科目。
- 北京市和山东省也确定要把 Python 编程基础纳入信息技术课程和高考的内容体系。
- 山东省最新出版的小学信息技术六年级教材中加入了 Python 内容。
- 浙江省信息技术教材将编程语言从 VB 更换为 Python。

这就是 Python 的火爆程度，中国已经开始全民学 Python 的时代了。未来 10~20 年后的年轻人，几乎人人都学过 Python，掌握编程变得就像如今会用 Office 般普遍。而对于现在的我们，掌握了 Python，就相当于跟上时代，掌握了未来……

一、本书特色

1. 大量的教学视频

为了让读者更好地学习本书，作者给每一章内容都录制了教学视频（一共 47 段，共 429 分钟）。借助这些视频，读者可以更轻松地学习。

2. 大量的典型应用实例，实战性强，有较高的应用价值

本书提供了 42 个 Python 相关的实战案例，理论讲解最终都落实到代码实现上。而且这些

案例会伴随着图书内容的推进，不断地趋近于工程化项目的风格，具有很高的应用价值和参考性。

3. 完整的源代码和配套素材

书中所有的代码都提供了免费下载，读者学习更方便。

另外，读者可以方便地获得书中案例的相关安装包和素材：如果是来源于网站的，则提供了有效下载链接；如果是作者制作的，则在随书资源中直接提供了。

4. 语法规则覆盖广

本书几乎囊括了 Python 3 版本中所涉及的全部语法规则，读者在系统学习之后，仍可将其当作一本 Python 语法工具书长伴左右，遇到生僻语法时及时查阅。

5. 商业案例，应用性强

本书提供的案例多数来源于真正的商业项目，具有高度的参考价值。有些代码甚至可以直接移植到自己的项目中，进行重复使用。使“从学到用”这个过程变得更加直接。

6. 大量宝贵经验的分享

授人以鱼，不如授之以渔。本书在讲解知识时，更注重方法与经验的传递。全书共有几十个“注意”标签，其中的内容都是含金量很高的成功经验分享与易错事项总结，有关于经验技巧的，有关于风险规避的，可以帮助读者在学习的路途上披荆斩棘，快速融会贯通。

二、本书读者对象

- Python 语言初学者
- Python 爬虫初学者
- Python 自动化运维初学者
- 人工智能初学者
- Python 开发工程师
- 人工智能开发工程师
- 使用 Python 进行数据分析的开发人员
- 需要提高动手能力的 Python 技术人员
- 大中专院校的相关学生

三、关于作者

本书由李金洪主笔编写，参与本书编写的还有以下作者。

兰世战

资深高级工程师，中国移动互联网技术专家，获得中国移动集团“互联网十佳技术能手”称号，在移动通信行业耕耘 13 年，现主要工作及兴趣为互联网大数据分析、AI、CDN 和区块链。主要使用 C、Python、R、Matlab、Java 等编程语言，先后从事传输网管、IP 网络、互联网、CDN、DNS 等专业和系统的研发、规划、建设和运维工作。已获得国家专利两项（独撰），集团和省级科技成果奖 13 项，发表学术论文 10 篇。

李昕

博士，副教授。2005 年起至今任教于北京邮电大学网络技术研究院“网络与交换技术国家重点实验室”，目前担任北京邮电大学教育部重点实验室副主任，中国互联网协会特聘青年专家。目前的研究方向包括：软件定义网络（SDN）、软件定义广域网（SDWAN）、天地一体化内容分发网络。

何建斌

在通信行业有 10 年以上的从业经验，其中 7 年在大数据相关技术领域。曾参与并主导多个数据平台的搭建。对 Hadoop 及 Spark 生态圈的技术栈有较深刻的认识。现任某知名互联网安全管理企业高级工程师，负责基于互联网数据的深度学习研发工作。

刘金成

长期从事互联网网络与信息安全工作，有着丰富的网络安全技术实战经验，成功预警和发现近几年发生的多起大规模网络安全事件。

现任某互联网安全管理中心工程师，负责基于深度学习的互联网安全事件监测与发现工作。

张晓宇

中国科学院信息工程研究所副研究员，硕士生导师。长期从事人工智能、模式识别领域前沿技术研究，“吴文俊人工智能科学技术创新奖”二等奖获得者，IEEE/ACM/CCF 核心成员、中国图像图形学会视觉大数据专委会副秘书长、中国自动化学会模式识别与机器智能专委会副秘书长。

许燕

一直致力于图像和音、视频领域的研究及应用，有多年一线写代码的经历。曾先后担任高级软件工程师、项目经理、技术经理、CTO 等职务。现担任某视频公司 CTO，全面负责产品研发及公司技术规划等。

眭新光

通信与信息系统博士，历任研发工程师、主任、项目经理、主管参谋，负责组织过大型国家专项项目、某大数据工程的规划论证。现任北京卓讯科信技术有限公司副总经理，负责战略规划和大型系统项目解决方案。

刘玉德

一直坚持在一线写代码。先后担任过高级软件工程师、项目经理、架构师、技术经理、CTO 等职务，曾经架构过某个大型的互联网金融平台。现任一家创业型农业公司 CTO，负责公司技术战略规划及团队建设。

另外，感谢本书的编辑吴宏伟先生，他为本书做了大量的细节调整。由于他的逐字推敲、一丝不苟，本书才变得语义更加通畅、内容更加通俗易懂。在此表示深深的感谢。

虽然我们对书中内容都进行了认真核实，并多次进行文字校对，但因时间和水平所限，书中疏漏和错误在所难免，敬请广者批评指正。联系作者请发 E-mail 到 94092670@qq.com，或者可以加入本书讨论 QQ 群：274962708。

如要联系编辑，请发 E-mail 到 wuhongwei@phei.com.cn。

李金洪

2018 年 4 月

目 录

第 1 篇 入门

| | |
|---|----|
| 第 1 章 了解 Python..... | 2 |
| 1.1 了解 Python 的历史..... | 2 |
| 1.2 Python 可以做什么..... | 3 |
| 1.3 Python 在数据科学中的地位及前景..... | 3 |
| 1.4 如何使用本书..... | 3 |
| 第 2 章 配置机器及搭建开发环境..... | 5 |
| 本章教学视频说明..... | 5 |
| 2.1 Python 版本及开发环境介绍..... | 6 |
| 2.1.1 Python 的运行方式..... | 6 |
| 2.1.2 常见的集成开发环境（IDE）..... | 7 |
| 2.2 下载及安装 Anaconda..... | 7 |
| 2.3 熟悉 Anaconda 3 开发工具..... | 10 |
| 2.3.1 快速了解 Spyder..... | 11 |
| 2.3.2 快速了解 Jupyter Notebook..... | 14 |
| 2.4 实例 1：运行 Python 程序，并传入参数..... | 14 |
| 2.4.1 在 Spyder 中新建 Python 文件，编写代码，运行代码..... | 15 |
| 2.4.2 用命令行启动 Python 程序，并传入参数..... | 16 |
| 2.4.3 用 Spyder 启动 Python 程序，并传入参数..... | 17 |
| 第 3 章 语言规则——Python 的条条框框..... | 19 |
| 本章教学视频说明..... | 19 |
| 3.1 了解编程语言的分类..... | 20 |
| 3.2 基础规则..... | 21 |
| 3.2.1 了解 Python 源代码相关的几个概念..... | 22 |
| 3.2.2 语句的基本规则：变量、语句、代码块..... | 22 |

- 3.2.3 添加注释 23
 - 3.2.4 使用 Python 的“帮助” 24
- 3.3 代码文件的结构 24
 - 3.3.1 模块 24
 - 3.3.2 包 25
- 3.4 模块的详细介绍 26
 - 3.4.1 模块的作用及分类 26
 - 3.4.2 模块的基本使用方法 27
 - 3.4.3 模块的搜索路径 27
 - 3.4.4 模块的属性 28
 - 3.4.5 模块名字的可变性 28
 - 3.4.6 模块的常规写法 29
- 3.5 模块的四种导入方式 29
 - 3.5.1 import as 方式 29
 - 3.5.2 from import 方式 29
 - 3.5.3 from import * 方式 30
 - 3.5.4 导入文件名中带空格的模块 31
- 3.6 实例 2：封装获取系统信息的模块，并将其导入 31
 - 3.6.1 在当前代码中的函数与模块中的函数同名情况下，导入模块 32
 - 3.6.2 在模块与当前代码不在同一路径的情况下，导入模块 34
 - 3.6.3 导入上级路径的模块 35

第 2 篇 进阶

- 第 4 章 变量——编写代码的基石 38
 - 本章教学视频说明 38
 - 4.1 什么是变量 39
 - 4.2 了解变量的规则 40
 - 4.2.1 明白变量的本质——对象 40
 - 4.2.2 同时定义多个变量 40
 - 4.2.3 变量类型介绍 41
 - 4.2.4 变量类型的帮助函数 41
 - 4.3 numbers（数字）类型 42

| | | |
|--------|--|----|
| 4.3.1 | 获取对象的类型 | 42 |
| 4.3.2 | 算术运算符 | 42 |
| 4.3.3 | 实例 3：演示“算术运算符”的使用 | 43 |
| 4.3.4 | 赋值运算符 | 44 |
| 4.3.5 | 实例 4：演示“赋值运算符”的使用 | 44 |
| 4.3.6 | 比较运算符 | 45 |
| 4.3.7 | 实例 5：演示“比较运算符”的使用 | 46 |
| 4.3.8 | 慎用 is 函数 | 48 |
| 4.3.9 | 实例 6：演示 Python 的缓存机制 | 50 |
| 4.3.10 | 布尔型关系的运算符 | 52 |
| 4.3.11 | 位运算符 | 53 |
| 4.3.12 | 实例 7：演示“位运算符”的使用 | 54 |
| 4.4 | strings（字符串）类型 | 54 |
| 4.4.1 | 字符串的描述 | 55 |
| 4.4.2 | 转义符 | 56 |
| 4.4.3 | 屏幕 I/O 及格式化 | 59 |
| 4.4.4 | 实例 8：以字符串为例，演示“序列”类型的运算及操作 | 65 |
| 4.4.5 | 关于切片的特殊说明 | 67 |
| 4.4.6 | 字符串的相关函数 | 68 |
| 4.5 | list（列表）类型 | 69 |
| 4.5.1 | list 的运算及操作 | 70 |
| 4.5.2 | list 的内置方法 | 70 |
| 4.5.3 | 实例 9：演示 list 使用中的技巧及注意事项 | 71 |
| 4.5.4 | 列表嵌套 | 74 |
| 4.5.5 | 实例 10：使用 list 类型实现队列和栈 | 74 |
| 4.5.6 | 实例 11：使用函数 filter 筛选列表——筛选学生列表中的偏科学生名单 | 77 |
| 4.6 | tuple（元组）类型 | 79 |
| 4.6.1 | tuple 的描述 | 79 |
| 4.6.2 | 运算及操作 | 80 |
| 4.6.3 | 实例 12：演示 tuple 的用法 | 80 |
| 4.7 | set（集合）类型 | 83 |
| 4.7.1 | set 的描述 | 83 |
| 4.7.2 | set 的运算 | 84 |

| | | |
|--------------|---------------------------------------|-----------|
| 4.7.3 | set 的内置方法 | 84 |
| 4.7.4 | 不可变集合 | 85 |
| 4.8 | dictionary（字典）类型 | 85 |
| 4.8.1 | 字典的描述 | 86 |
| 4.8.2 | 字典的运算 | 86 |
| 4.8.3 | 字典的内置方法 | 88 |
| 4.9 | 对组合对象进行“深拷贝”和“浅拷贝” | 88 |
| 4.9.1 | 浅拷贝 | 88 |
| 4.9.2 | 深拷贝 | 89 |
| 第 5 章 | 控制流——控制执行顺序的开关 | 91 |
| | 本章教学视频说明 | 91 |
| 5.1 | if 语句 | 92 |
| 5.1.1 | 语句形式 | 92 |
| 5.1.2 | 演示 if 语句的使用 | 92 |
| 5.1.3 | 实例 13：根据来访人的性别选择合适的称呼 | 93 |
| 5.2 | while 语句 | 94 |
| 5.2.1 | 语句形式 | 94 |
| 5.2.2 | 演示 while 语句的使用 | 94 |
| 5.2.3 | 实例 14：将十进制数转化为二进制数 | 95 |
| 5.3 | for 语句 | 96 |
| 5.3.1 | 语句形式 | 96 |
| 5.3.2 | 在 for 循环中，使用切片 | 96 |
| 5.3.3 | 在 for 循环中，使用内置函数 range | 97 |
| 5.3.4 | 实例 15：利用循环实现冒泡排序 | 98 |
| 5.3.5 | 在 for 循环中，使用内置函数 zip | 99 |
| 5.3.6 | 在 for 循环中，使用内置函数 enumerate | 101 |
| 5.4 | 对循环进行控制——break、continue、pass 语句 | 102 |
| 5.5 | 实例 16：演示人机对话中的控制流程（综合应用前面语句） | 102 |
| 5.6 | 利用 for 循环实现列表推导式 | 104 |
| 5.7 | 实例 17：利用循环来打印“九九乘法表” | 105 |
| 5.8 | 理解 for 循环的原理——迭代器 | 106 |

| | |
|---|-----|
| 第 6 章 函数——功能化程序片段的封装..... | 108 |
| 本章教学视频说明 | 108 |
| 6.1 函数的基本概念 | 109 |
| 6.1.1 函数的定义 | 109 |
| 6.1.2 函数的组成部分 | 109 |
| 6.1.3 函数的参数：形参与实参 | 110 |
| 6.1.4 函数的返回值 | 111 |
| 6.1.5 函数的属性 | 111 |
| 6.1.6 函数的本质 | 112 |
| 6.1.7 函数的分类 | 112 |
| 6.1.8 实例 18：打印两个心形图案 | 113 |
| 6.2 定义参数及调用函数 | 115 |
| 6.2.1 函数参数的定义方法与调用形式..... | 115 |
| 6.2.2 在函数调用中检查参数 | 121 |
| 6.2.3 函数调用中的参数传递及影响..... | 122 |
| 6.3 匿名函数与可迭代函数 | 124 |
| 6.3.1 匿名函数与可迭代函数的介绍..... | 124 |
| 6.3.2 匿名函数与 reduce 函数的组合应用 | 125 |
| 6.3.3 匿名函数与 map 函数的组合应用..... | 125 |
| 6.3.4 匿名函数与 filter 函数的组合应用 | 126 |
| 6.3.5 可迭代函数的返回值 | 127 |
| 6.4 偏函数 | 128 |
| 6.5 递归函数 | 129 |
| 6.6 eval 与 exec 函数 | 130 |
| 6.6.1 eval 与 exec 的区别 | 130 |
| 6.6.2 eval 与 exec 的定义 | 130 |
| 6.6.3 exec 和 eval 的使用经验 | 132 |
| 6.6.4 eval 与 exec 的扩展知识 | 134 |
| 6.7 实例 19：批量测试转化函数（实现“组合对象”与“字符串”的相互转化） | 134 |
| 6.7.1 编写两个功能函数 | 135 |
| 6.7.2 编写单元测试用例 | 135 |
| 6.7.3 批量运行单元测试用例 | 136 |

| | | |
|--------|--------------------------|-----|
| 6.8 | 生成器函数 | 137 |
| 6.8.1 | 生成器与迭代器的区别 | 137 |
| 6.8.2 | 生成器函数 | 137 |
| 6.8.3 | 生成器表达式 | 137 |
| 6.9 | 变量的作用域 | 138 |
| 6.9.1 | 作用域介绍 | 138 |
| 6.9.2 | global 语句 | 140 |
| 6.9.3 | nonlocal 语句 | 140 |
| 6.10 | 工厂函数 | 141 |
| 6.10.1 | 普通工厂函数的实现 | 141 |
| 6.10.2 | 闭合函数 (closure) | 142 |
| 6.10.3 | 装饰器 (decorator) | 143 |
| 6.10.4 | @修饰符 | 144 |
| 6.10.5 | 更高级的装饰器 | 145 |
| 6.10.6 | 解决“同作用域下默认参数被覆盖”问题 | 151 |
| 第 7 章 | 错误与异常——调教出听话的程序 | 153 |
| | 本章教学视频说明 | 153 |
| 7.1 | 错误的分类 | 154 |
| 7.1.1 | 语法错误 | 154 |
| 7.1.2 | 运行时错误 | 154 |
| 7.2 | 异常的基本语法 | 155 |
| 7.2.1 | 同时处理多个异常 | 156 |
| 7.2.2 | 异常处理中的 else 语句 | 157 |
| 7.2.3 | 输出未知异常 | 157 |
| 7.2.4 | 输出异常的详细信息 | 158 |
| 7.3 | 捕获与处理异常 | 160 |
| 7.3.1 | 异常的处理流程 | 161 |
| 7.3.2 | try 语句的工作原理 | 161 |
| 7.3.3 | 一些常见的异常 | 161 |
| 7.4 | 创建异常 | 163 |
| 7.4.1 | 创建异常的方法 | 163 |
| 7.4.2 | 创建异常举例 | 163 |

| | | |
|-------|---|-----|
| 7.5 | 异常的最终处理（清理动作） | 164 |
| 7.5.1 | finally 的使用场景 | 164 |
| 7.5.2 | finally 与 else 的区别 | 165 |
| 7.6 | 判定条件的正确性（断言） | 165 |
| 7.6.1 | 断言的表达形式 | 165 |
| 7.6.2 | 带错误信息的断言语句 | 166 |
| 7.7 | 实例 20：如 HTTP 请求失败，实现“重试”功能 | 166 |
| 7.7.1 | 使用装饰器实现失败重试 | 167 |
| 7.7.2 | 编写简单爬虫 | 168 |
| 7.7.3 | 传入正确的目的地址，开始爬取 | 168 |
| 7.7.4 | 传入错误的目的地址，验证“重试”功能 | 169 |
| 第 8 章 | 文件操作——数据持久化的一种方法 | 170 |
| | 本章教学视频说明 | 170 |
| 8.1 | 文件的基本操作 | 171 |
| 8.1.1 | 读写文件的一般步骤 | 171 |
| 8.1.2 | 打开文件 | 172 |
| 8.1.3 | 具体读写 | 173 |
| 8.1.4 | 关闭文件 | 174 |
| 8.2 | 文件对象的方法 | 175 |
| 8.2.1 | 文件对象的常用方法介绍 | 175 |
| 8.2.2 | 把文件对象当作迭代器来读取 | 176 |
| 8.3 | 实例 21：带有异常处理的文件操作 | 177 |
| 8.4 | 使用 with 语句简化代码 | 178 |
| 8.4.1 | 实例 22：使用 with 语句操作文件 | 178 |
| 8.4.2 | with 语法的原理 | 179 |
| 8.5 | 实现字符串与二进制数的相互转化 | 179 |
| 8.5.1 | 将字符串转二进制数 | 180 |
| 8.5.2 | 将二进制数转字符串 | 180 |
| 8.6 | 将任意对象序列化 | 181 |
| 8.6.1 | pickle 函数 | 181 |
| 8.6.2 | 实例 23：用 pickle 函数实现元组与“二进制对象”“二进制对象文件”之间的转换 | 183 |

| | |
|-----------------------------------|-----|
| 8.6.3 序列化的扩展方法（ZODB 模块） | 185 |
| 8.7 实例 24：批量读取及显示 CT 医疗影像数据 | 185 |
| 8.7.1 DICOM 介绍 | 185 |
| 8.7.2 Python 中的 DICOM 接口模块 | 186 |
| 8.7.3 编写代码以载入 DICOM 文件 | 186 |
| 8.7.4 读取 DICOM 中的数值 | 187 |
| 8.7.5 显示单张 DICOM 数据图像 | 189 |
| 8.7.6 批量生成 DICOM 数据图像 | 189 |

第 3 篇 高阶

| | |
|---|-----|
| 第 9 章 类——面向对象的编程方案 | 192 |
| 本章教学视频说明 | 192 |
| 9.1 类的相关术语及实现 | 193 |
| 9.1.1 创建类 | 194 |
| 9.1.2 创建类属性 | 194 |
| 9.1.3 定义类的动态属性 | 195 |
| 9.1.4 限制类属性（__slots__） | 196 |
| 9.2 实例化类对象 | 197 |
| 9.2.1 带有初始值的实例化 | 197 |
| 9.2.2 class 中的 self | 198 |
| 9.2.3 类方法（@classmethod）与静态方法（@staticmethod） | 200 |
| 9.2.4 类变量与实例变量的区别 | 202 |
| 9.2.5 销毁类实例化对象 | 203 |
| 9.3 类变量的私有化类属性 | 204 |
| 9.3.1 公有化（public）与私有化（private） | 204 |
| 9.3.2 私有化的实现 | 205 |
| 9.3.3 使用装饰器技术实现类的私有化（@property） | 207 |
| 9.4 实现子类 | 209 |
| 9.4.1 继承 | 209 |
| 9.4.2 实例 25：演示类的继承 | 211 |
| 9.4.3 super 函数 | 212 |
| 9.4.4 实例 26：演示 super 函数的功能 | 212 |

| | | |
|--------|--------------------------------------|-----|
| 9.5 | 类相关的常用内置函数 | 216 |
| 9.5.1 | 判断实例 (isinstance) | 216 |
| 9.5.2 | 判断子类 (issubclass) | 216 |
| 9.5.3 | 判断类实例中是否含有某个属性 (hasattr) | 217 |
| 9.5.4 | 获得类实例中的某个属性 (getattr) | 217 |
| 9.5.5 | 设置类实例中的某个属性值 (setattr) | 217 |
| 9.6 | 重载运算符 | 218 |
| 9.6.1 | 重载运算符的方法与演示 | 218 |
| 9.6.2 | 可重载的运算符 | 219 |
| 9.7 | 包装与代理 | 220 |
| 9.7.1 | 包装 | 220 |
| 9.7.2 | 代理 | 221 |
| 9.7.3 | 实例 27：使用代理的方式实现 RESTful API 接口 | 222 |
| 9.8 | 自定义异常类 | 225 |
| 9.8.1 | 自定义异常类的方法 | 225 |
| 9.8.2 | 实例 28：自定义异常类的多重继承与使用 | 226 |
| 9.9 | 支持 with 语法的自定义类 | 228 |
| 9.9.1 | 实现支持 with 语法的类 | 229 |
| 9.9.2 | 实例 29：代码实现自定义类，并使其支持 with 语法 | 229 |
| 9.10 | “自定义迭代器类”的实现与调试技巧 | 231 |
| 9.10.1 | 实例 30：自定义迭代器，实现字符串逆序 | 231 |
| 9.10.2 | 调试技巧 | 233 |
| 9.11 | 元类 (MetaClass) | 238 |
| 9.11.1 | Class 的实现原理 | 238 |
| 9.11.2 | 元类的介绍 | 239 |
| 第 10 章 | 系统调度——实现高并发的处理任务 | 242 |
| | 本章教学视频说明 | 242 |
| 10.1 | 进程与线程 | 243 |
| 10.2 | 线程 | 243 |
| 10.2.1 | 线程的创建及原理 | 244 |
| 10.2.2 | 互斥锁 | 248 |
| 10.2.3 | 实例 31：使用信号量来同步多线程间的顺序关系 | 251 |

| | | |
|--------|-----------------------------------|-----|
| 10.2.4 | 实例 32: 实现基于事件机制的消息队列..... | 254 |
| 10.2.5 | 实例 33: 使用条件锁同步多线程中的生产者与消费者关系..... | 257 |
| 10.2.6 | 实例 34: 创建定时器触发程序, 在屏幕上输出消息..... | 259 |
| 10.2.7 | 实例 35: 使用线程池提升运行效率..... | 261 |
| 10.2.8 | 需要创建多少个线程才算合理..... | 265 |
| 10.3 | 进程..... | 265 |
| 10.3.1 | 实例 36: 创建多进程的程序..... | 265 |
| 10.3.2 | 多进程与多线程的区别..... | 268 |
| 10.4 | 协程..... | 268 |
| 10.4.1 | 协程的相关概念及实现步骤..... | 269 |
| 10.4.2 | 实例 37: 使用协程实现“任务提交”与“结果接收”..... | 270 |
| 10.5 | 应该选择线程, 还是协程..... | 271 |
| 10.6 | 实例 38: 使用协程批量修改文件扩展名..... | 271 |

第 4 篇 商业实战

| | | |
|--------|---|-----|
| 第 11 章 | 爬虫实战 (实例 39): 批量采集股票数据, 并保存到 Excel 中..... | 276 |
| 11.1 | 爬取股票代码..... | 276 |
| 11.1.1 | 找到目标网站..... | 277 |
| 11.1.2 | 打开调试窗口, 查看网页代码..... | 277 |
| 11.1.3 | 在网页源码中找到目标元素..... | 278 |
| 11.1.4 | 分析目标源代码, 找出规律..... | 278 |
| 11.1.5 | 编写代码抓取股票代码..... | 279 |
| 11.1.6 | 运行代码, 显示结果..... | 280 |
| 11.2 | 爬取股票内容..... | 280 |
| 11.2.1 | 编写代码抓取批量内容..... | 280 |
| 11.2.2 | 运行代码显示结果..... | 281 |
| 11.3 | 爬虫项目的其他技术..... | 282 |
| 第 12 章 | 自动化实战 (实例 40): 读取 Excel 数据文件, 并用可视化分析..... | 283 |
| 12.1 | 使用 Pandas 读取 Excel 文件, 并用 Matplotlib 生成大盘走势图..... | 283 |
| 12.2 | 使用 Pandas 处理数据并绘制成图..... | 285 |
| 12.2.1 | 使用 Pandas 处理数据..... | 285 |

| | | |
|--------|--|-----|
| 12.2.2 | 绘制直方图和折线图 | 286 |
| 12.3 | 自动化数据处理的其他技术 | 287 |
| 第 13 章 | 机器学习实战（实例 41）：从一组看似混乱的数据中找出 $y \approx 2x$ 的规律 | 288 |
| 13.1 | 准备数据 | 288 |
| 13.2 | 训练模型并实现可视化 | 290 |
| 13.2.1 | 训练模型 | 290 |
| 13.2.2 | 利用模型进行预测 | 290 |
| 13.2.3 | 了解线性回归模型的内部原理 | 291 |
| 13.2.4 | 将模型可视化 | 291 |
| 13.3 | 评估模型 | 292 |
| 13.3.1 | 评估模型的基本思路 | 292 |
| 13.3.2 | 用代码实现模型评估 | 293 |
| 13.4 | 保存模型，应用模型 | 293 |
| 13.5 | 机器学习的方向 | 294 |
| 第 14 章 | 人工智能实战（实例 42）：基于人脸识别的“来访登记系统” | 295 |
| 14.1 | 安装案例所依赖的模块 | 295 |
| 14.2 | 安装及使用 dlib 模块 | 296 |
| 14.2.1 | 下载 dlib 模块 | 296 |
| 14.2.2 | 安装 dlib 模块 | 298 |
| 14.2.3 | 使用 dlib 模块进行人脸检测 | 299 |
| 14.3 | 安装及使用 face_recognition 模块 | 300 |
| 14.3.1 | 下载 face_recognition 模块 | 300 |
| 14.3.2 | 下载及安装 face_recognition_models 模型 | 301 |
| 14.3.3 | 使用 face_recognition 模块检测人脸中的特征点 | 303 |
| 14.4 | 安装及使用 opencv 模块 | 304 |
| 14.4.1 | 下载并安装 opencv 模块 | 304 |
| 14.4.2 | 下载中文字体 | 304 |
| 14.4.3 | 使用 opencv 调用摄像头进行拍照 | 305 |
| 14.5 | 安装及使用 yagmail 模块 | 306 |
| 14.5.1 | 安装 yagmail 模块 | 306 |
| 14.5.2 | 使用 yagmail 模块向自己的 QQ 邮箱发送邮件 | 306 |

| | | |
|--------|--------------------------|-----|
| 14.6 | 详细设计 | 309 |
| 14.6.1 | 需求描述 | 310 |
| 14.6.2 | 定义系统的输入和输出 | 310 |
| 14.6.3 | 系统规则及约束 | 310 |
| 14.6.4 | 结构体设计 | 310 |
| 14.6.5 | 软件的主体架构图 | 311 |
| 14.6.6 | 软件的主体流程介绍 | 312 |
| 14.7 | 编码实现 | 312 |
| 14.7.1 | 导入模块 | 312 |
| 14.7.2 | 定义结构体 | 313 |
| 14.7.3 | 实现发送邮件函数 | 313 |
| 14.7.4 | 实现邮件内容生成函数 | 313 |
| 14.7.5 | 实现过滤并保存来访记录的函数 | 314 |
| 14.7.6 | 实现定时器处理函数 | 314 |
| 14.7.7 | 在主线程中启动定时器线程 | 315 |
| 14.7.8 | 实现并调用函数载入人脸库 | 315 |
| 14.7.9 | 在主循环里调用摄像头，并进行人脸识别 | 316 |
| 14.8 | 运行程序 | 318 |
| 14.9 | 下一步对系统的改进 | 319 |
| 附录 A | 内置函数 | 321 |

第 1 篇 入门

本篇介绍了 Python 语言的发源及特点、如何搭建 Python 开发环境、Python 的语言规则，并演示了如何启动 Python 程序。一方面使读者对 Python 这门编程语言产生熟悉的感觉；另一方面也对它的基础规则有一个大体的了解，为后面的学习做准备。

- ▶ 第 1 章 了解 Python
- ▶ 第 2 章 配置机器及搭建开发环境
- ▶ 第 3 章 语言规则——Python 的条条框框

第 1 章

了解Python

Python 是一门强大的解释型、面向对象的高级程序设计语言，它优雅、简单、可移植、易扩展，可用于桌面应用、系统编程、数据库编程、网络编程、Web 开发、图像处理、人工智能、数学应用、文本处理等。

1.1 了解 Python 的历史

Python 的几个重大事件如下：

1989 年，Guido van Rossum 开发了 Python 语言。

1999 年，第一个基于 Python 的 Web 框架（ZOEPE）诞生。

2000 年，Python 2.0 版本发布，其中加入了内存回收机制，构成了现在 Python 语言框架的基础。

2004 年，Python 2.4 版本诞生，同时也诞生了 Django 这个 Web 框架。

2008 年，Python 3.0 版本诞生，其在 2.X 版本上实现了一次大的跳跃。

2010 年，诞生了目前应用最广泛的 Python 2.7 版本。它是一个承接 2.X 和 3.X 特性的过渡版本。

目前，官方宣布 Python 2.X 系列将在 2020 年结束支持。从长远来看，Python 3.X 会有更大的发展空间。

1.2 Python 可以做什么

Python 是一门真正意义上的编程语言，除了可用于编写脚本、数值计算外，还可以用于编写命令行程序、编写带用户界面的应用程序、编写网站、绘制图形等。

使用 Python，可以很容易地编写跨平台的应用程序。这些应用程序可以在不同的平台上直接使用，也可以被编译为独立的程序运行。Python 脚本可以用来驱动 MySQL、Sqlite、ArcGIS、Adobe Indesign 等软件，也可以调用 C、C++ 以及 Fortran 等语言的函数库，用途极广。

Python 的应用领域非常广阔。很多人学习 Python，不但是为了掌握一门技术，更是为了拥有更多的资源。GitHub 上提供了数以万计的 Python 开源项目，供爱好者仔细研究。掌握了 Python，就可以拥有并掌握这些资源。

1.3 Python 在数据科学中的地位及前景

目前，Python 已经成为人工智能的第一主流语言。市面上各大主流人工智能框架——TensorFlow、CNTK、Caffe2 等，都支持用户使用 Python 语言进行开发。

Python 的语法简洁、通俗易懂，很容易上手。Python 用户不需要使用层次复杂的花括号，只需留意缩进。Python 还有完善的程序包管理系统（pip 命令），使得安装与编写程序包都极为方便。

Python 语言的学习，已经上升到了国家战略的层面。国家相关教育部门对“人工智能的普及”格外重视，将 Python 列入小学、中学和高中的教育体系中，为未来国家和社会的发展奠定了人工智能的人才基础，逐步由底层向高层推动“全民学 Python”，从而进一步推动人工智能技术和社会人才结构的更迭。

关于人工智能技术，推荐读者阅读作者的另一本书——《深度学习之 TensorFlow：入门、原理与进阶实战》。

1.4 如何使用本书

本书配有相关视频可供读者参考。针对不同基础的读者，作者给出如下学习建议：

1. 没有基础，习惯于听课式学习的读者

对于这类读者，建议直接去看本书的配套视频，同时对照书中的章节做好标注。对于习惯听课式学习的读者，视频可以充当讲课的老师，书籍可以充当课堂笔记。

2. 有一定基础，想提升自己的读者

对于这类读者，如果时间有限，建议先看目录，直接找到自己关心的技术点去详细阅读；如果时间允许，还是建议把这本书从头到尾看一遍，这样可以系统而全面地掌握 Python 知识。

3. 已经掌握 Python 语法知识的读者

对于已经掌握 Python 语法知识，或是已经学完本书内容的读者，本书还可以作为一个工具书，充当“语法字典”。

在众多编程语言中，Python 的语法还是有其独特性的。有些地方很有“说道”。对于一些不常使用的语法规则，作者本人有时也难免会有遗忘（出于工作需要，很多时候作者需要同时使用好几种编程语言编写代码，难免会弄混）。尤其在学习他人的开源代码过程中，当遇到某些语法没有看懂时，可以即时查阅本书。并不是所有的知识点都可以轻松地在网上查到，有些不常用的知识点找起来会很费事。在身边备上本书，可以即时查阅，会很省时间。

第 2 章

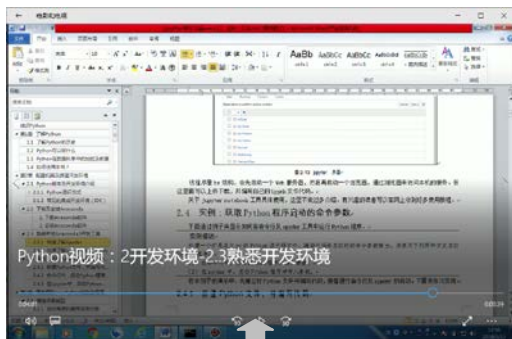
配置机器及搭建开发环境

本章主要服务于零基础的学员，介绍了如何在 Windows 操作系统中搭建 Python 3.0 版本的开发环境。示例中的开发工具为 Anaconda。

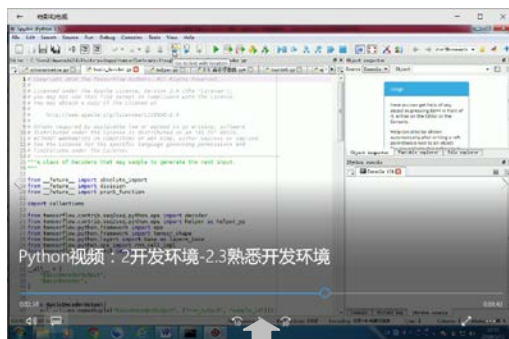
当然，读者也可以根据自己的习惯使用其他操作系统（如 Linux、Mac）或是别的开发工具。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解



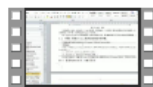
具体代码操作演示



Python视频：
2开发环境
-2.1-2.2版本及
安装.mp4



Python视频：
2开发环境-2.3
熟悉开发环境.
mp4



Python视频：
2开发环境-2.4
实例1演示程
序.mp4

本章共有 3 段教学视频，总时长为 11 min 左右。包含以下内容：

- 讲述 Python 的版本及开发环境的安装。
- 介绍了 Anaconda 的开发环境。
- 通过对书中 2.4 节实例的演示来讲解 Spyder 的使用。

2.1 Python 版本及开发环境介绍

Python 的版本大致可以分为两类—— Python 3.0 以前的版本和 Python 3.0 以后的版本。这两类版本的语法差异比较大，且兼容性较差。

- Python 3.0 以后的版本，从语法设计到具体实现，都解决了很多之前版本中所出现的问题与缺陷，是当今的主流版本。在 Python 3.0 以后的版本中，相对比较新的是 3.6 版本。
- 在 Python 3.0 以前的版本中，相对比较经典的是 2.7 版本。

本书以 Python 3.6 版本来讲解 Python。

2.1.1 Python 的运行方式

早先的 Python，都是使用命令行的方式来运行的。后来，又出现了很多带有界面的开发工具，可以交互的方式运行 Python。下面依次介绍。

1. 命令行方式

命令方式是一种最基础方式。直接在 DOS 窗口（如图 2-1 所示）中输入具体的命令，以运行 Python 程序。由于 Python 是解释性语言，也可以在命令行里一句一句地输入命令。一边编程，一边执行。

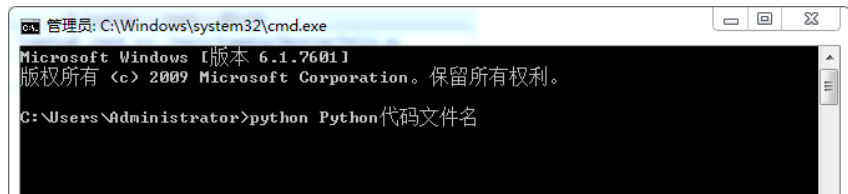


图 2-1 命令行界面

用户可以在命令行窗口下，通过“python+空格+文件名”的方式来运行一个 Python 代码文件，如图 2-1 所示。（只有在本机安装好 Python 环境后，该命令才会生效）

2. 图形交互式

图形交互式，是一种相对高级的运行方式。在调试和编写代码时，用户感觉更加友好。后面的 2.3.1 小节会介绍操作界面。

一般来讲，编写代码时最好使用图形交互式。待代码成熟后，运行应用可以使用命令行的方式。命令行方式，可以使程序在运行时节省不必要的开销。

2.1.2 常见的集成开发环境（IDE）

关于 Python 的集成开发环境有很多，如 Upterm、Ptpython、Sublime3、IPython 等。本书重点介绍 Anaconda 开发环境。使用 Anaconda 的好处是集成性高。Anaconda 中包含了很多常用的开发软件包，用户无需再下载和配置 Python 的各种安装包，省去了大量的时间。

2.2 下载及安装 Anaconda

在 Anaconda 的环境搭建中，重点是版本的选择。下面来详细介绍下 Anaconda 的下载及安装。

1. 下载 Anaconda 软件

(1) 通过 <https://www.anaconda.com> 来到 Anaconda 官网。

(2) 单击右上角的 Download 按钮，如图 2-2 所示。



图 2-2 Anaconda 首页

(3) 将网页拉到下方，单击界面最右侧的链接“Packages Included in Anaconda”，如图 2-3 所示。

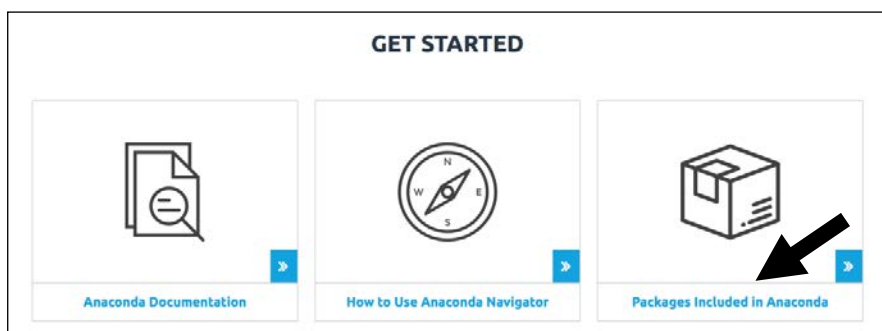


图 2-3 单击 “Packages Included in Anaconda” 选项

(4) 进入 Packages Included in Anaconda 页，单击图中最后一行的 “package repository”，如图 2-4 所示。



图 2-4 单击 “package repository”

(5) 进入 Anaconda repository 页面，如图 2-5 所示。单击图中的“Anaconda installer archive”，下载完全版本。

Anaconda repositories

Updates are available as an [RSS feed](#).
Anaconda is brought to you by [Anaconda, Inc.](#)
[Anaconda installer archive](#) [download page](#). ([winzip](#))
Miniconda installers [download page](#). ([winzip](#))
Please also, check out [anaconda.org](#).

图 2-5 下载连接

(6) 完全版本的安装文件如图 2-6 所示。其中有 Linux、Windows、MacOSX 的各种版本，可以根据需要自行选择。

| Anaconda installer archive | | | | |
|------------------------------------|--------|---------------------|-----------------------------------|--|
| Filename | Size | Last Modified | MD5 | |
| Anaconda2-5.1.0-Linux-ppc64le.sh | 267.3M | 2018-02-15 09:08:49 | e894dcc547alc7d67deb04f6bba7223a | |
| Anaconda2-5.1.0-Linux-x86.sh | 431.3M | 2018-02-15 09:08:51 | e26fb9d3e53049f6e32212270af6b987 | |
| Anaconda2-5.1.0-Linux-x86_64.sh | 533.0M | 2018-02-15 09:08:50 | 5b1b5784cae93cf696e11e66983d8756 | |
| Anaconda2-5.1.0-MacOSX-x86_64.pkg | 588.0M | 2018-02-15 09:08:52 | 4f9c197dfe6d3dc7e50a8611b4d3cfa2 | |
| Anaconda2-5.1.0-MacOSX-x86_64.sh | 505.9M | 2018-02-15 09:08:53 | e9845ccf67542523c5be09552311666e | |
| Anaconda2-5.1.0-Windows-x86.exe | 419.8M | 2018-02-15 09:08:55 | a09347a53e04a15ee965300c2b95dfde | |
| Anaconda2-5.1.0-Windows-x86_64.exe | 522.6M | 2018-02-15 09:08:54 | b16d6d6858fc7decf671ac71e6d7cfd8 | |
| Anaconda3-5.1.0-Linux-ppc64le.sh | 285.7M | 2018-02-15 09:08:56 | 47b5b2b17b7dbac0d4d0f0a4653f5b1c | |
| Anaconda3-5.1.0-Linux-x86.sh | 419.7M | 2018-02-15 09:08:58 | 793a94ee85baf64d0ebb67a0c49af4d7 | |
| Anaconda3-5.1.0-Linux-x86_64.sh | 551.2M | 2018-02-15 09:08:57 | 966406059cf7ed89cc82eb475ba506e5 | |
| Anaconda3-5.1.0-MacOSX-x86_64.pkg | 594.7M | 2018-02-15 09:09:06 | 6ed496221b843d1b5fe8463d3136b649 | |
| Anaconda3-5.1.0-MacOSX-x86_64.sh | 511.3M | 2018-02-15 09:10:24 | 047e12523fd287149ecd80c803598429 | |
| Anaconda3-5.1.0-Windows-x86.exe | 435.5M | 2018-02-15 09:10:28 | 7a2291ab99178a4dec530861494531f | |
| Anaconda3-5.1.0-Windows-x86_64.exe | 537.1M | 2018-02-15 09:10:26 | 83a8b1edeb21fa0ac481b23f65b604c6 | |
| Anaconda2-5.0.1-Linux-x86.sh | 413.2M | 2017-10-24 12:13:07 | ae155b192027e23189d723a897782fa3 | |
| Anaconda2-5.0.1-Linux-x86_64.sh | 507.7M | 2017-10-24 12:13:52 | dc13fe5502cd78dd03e8a727bb9be63f | |
| Anaconda2-5.0.1-Windows-x86.exe | 403.4M | 2017-10-24 12:08:14 | 623e8d9ca2270cb9823a897dd0e9bfc | |
| Anaconda3-5.0.1-Windows-x86.exe | 420.4M | 2017-10-24 12:37:10 | 9d2ffb0aac1f8a72ef4a5c535f3891f2 | |
| Anaconda3-5.0.1-Windows-x86_64.exe | 514.8M | 2017-10-24 12:37:59 | 3dde7dbbef158db6dc44fce495671c92 | |
| Anaconda2-5.0.1-MacOSX-x86_64.pkg | 562.8M | 2017-10-23 20:01:12 | 46fc99d1cfe1e27f3b2a3eb63fee1a532 | |
| Anaconda2-5.0.1-MacOSX-x86_64.sh | 486.5M | 2017-10-23 19:51:04 | 17314016dced36614a3bef8ff3db7056 | |
| Anaconda2-5.0.1-Windows-x86_64.exe | 499.8M | 2017-10-23 21:57:22 | b8d9bc02edd61af37fce3d07e726e91 | |

图 2-6 下载列表（部分）

本书中使用的是 Windows 64 位下的 Python 3.6 版本，选择对应的安装包为 Anaconda3-5.0.1-Windows-x86_64.exe（见图 2-6 中的标注）。



提示：

本书的内容均是基于 Python 3.6 版本。

虽然 Python 3 以上的版本算作同一阶段，但是每个版本间也会略有区别（例如：Python 3.5 与 Python 3.6），并且没有向下兼容。在与其他的 Python 软件包整合使用时，一定要按照需要整合软件包的说明文件来找到完全匹配的 Python 版本，否则会带来不可预料的麻烦。

另外，不同版本的 Anaconda 默认支持的 Python 版本是不一样的：支持 Python 2 的版本 Anaconda，统一以“Anaconda 2”为开头来命名；支持 Python 3 的版本 Anaconda，统一以“Anaconda 3”为开头来命名。Anaconda 当前最新的版本为 5.1.0，可以支持 Python 3.6 版本。

2. 安装 Anaconda 软件

在 Windows 下安装 Anaconda 软件的方法，与一般的软件安装相似。右击安装包，在弹出的快捷菜单中选择“以管理员身份运行”命令即可。然后根据提示指定安装的路径。这里假设安装路径为 C:\local\Anaconda。

在安装期间，会出现注册环境变量的页面，如图 2-7 所示。

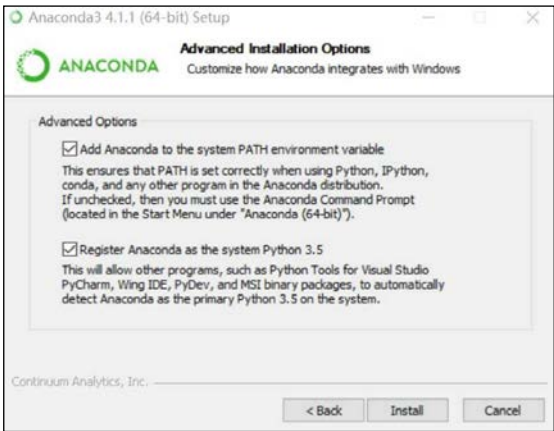


图 2-7 注册环境变量的页面

图 2-7 中有两个复选框，建议全部都勾选，表示要注册环境变量。只有注册好环境变量，才可以在命令行下通过 `Python` 命令运行程序。

安装好 Anaconda 后，与 Python 配套的常用第三方库也一并安装好。其所在的路径如下：

```
C:\local\Anaconda3\Lib\site-packages
```

如果想要再安装其他的第三方库，可以使用 Anaconda 中自带的 `pip` 软件，即在命令行下直接输入“`pip+空格+第三方安装包名称`”即可。运行 `pip` 命令之后，系统会自动从网上下载相关的安装包，并安装到本机。例如，下面是在本机上安装深度学习框架 TensorFlow 的命令：

```
C:\Users\Administrator>pip install tensorflow
```

如果要卸载某个第三方安装包，直接将上面例子的 `install` 替换成 `uninstall` 即可。

2.3 熟悉 Anaconda 3 开发工具

在本书中使用到的开发环境是 Anaconda 3。在 Anaconda 3 里一般常用的有两个工具：Spyder、Jupyter Notebook，它们在“开始”菜单下的 Anaconda 3（64-bit）目录下，如图 2-8 所示。

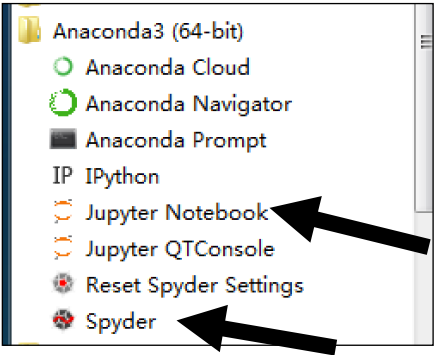


图 2-8 两个常用工具的位置

2.3.1 快速了解 Spyder

本书推荐使用 Spyder 作为编译器的原因是：它比较方便，属于 Anaconda 安装包中自带的工具，不需要再额外安装其他东西，省去了大量的搭建环境时间；Spyder 的 IDE 功能也很强大，基本上可以满足日常需要。

下面通过几个常用的功能来介绍其使用细节。

1. 面板介绍

Spyder 主界面可以分为 7 个区域，如图 2-9 所示。

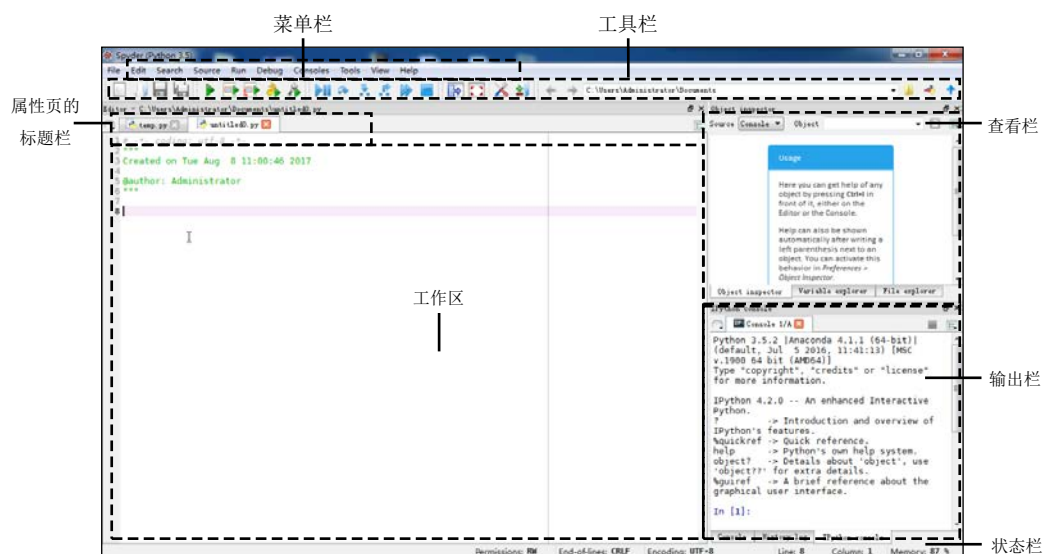


图 2-9 Spyder 面板

- (1) 菜单栏：其中包含软件中所支持的全部功能；
- (2) 工具栏：是菜单栏的快捷方式。具体放置哪些工具，可以通过勾选菜单“View /Toolbar”里的命令来实现，如图 2-10 所示；
- (3) 工作区：编写代码的地方；
- (4) 属性页的标题栏：用于显示当前代码的名字及位置；

注“2”的按钮为“运行设置”按钮，单击该按钮会弹出“Run Settings”窗口，在其中可以设置输入启动程序的参数。

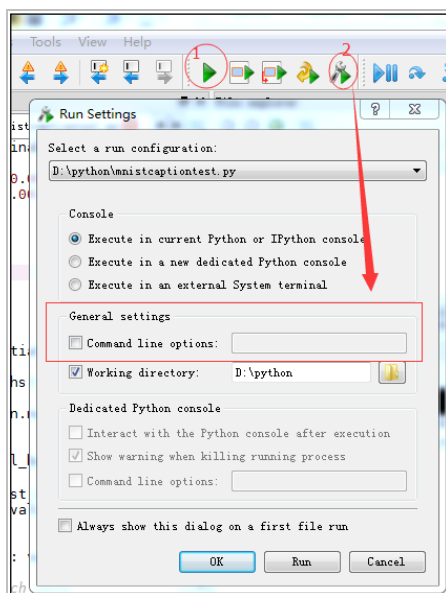



图 2-12 运行相关的按钮

4. 调试功能

图 2-12 中右侧的  按钮为调试功能的按钮。在 Python 运行时，可以通过设置断点来进行调试。

5. 源码操作工具条

当同时打开多个代码时，可能希望回到刚才看的代码的位置。在 Spyder 中，有一个功能可以帮你实现。在图 2-10 中勾选“Source toolbar”，会在快捷菜单栏看到如图 2-13 所示的几个图标。从左边起，第一个为建立书签，第二个为回退到上次代码位置，第三个为前进到下次代码位置。



图 2-13 Source

以上都是 Spyder 的常用操作。Spyder 还有很多功能，这里就不一一介绍了。

2.3.2 快速了解 Jupyter Notebook

在网络上看到的一些深度学习源码中，好多扩展名为 `ipynb`。这种扩展名为 `ipynb` 的文件是使用 Jupyter Notebook 软件生成的。

Jupyter Notebook 是一个界面非常友好的代码开发工具。使用 Jupyter Notebook 打开的代码，既可以作为说明文档，又可以作为可运行的 Python 代码文件。Anaconda 中也集成了这个软件。

在图 2-8 所示位置找到“Jupyter Notebook”并单击，看到如图 2-14 界面。

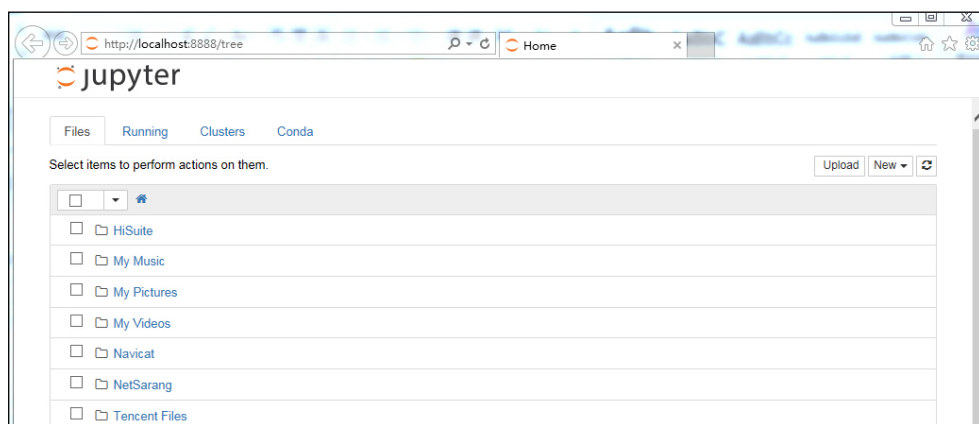


图 2-14 Jupyter Notebook 界面

Jupyter Notebook 是 B/S 结构，会先启动一个 Web 服务器，然后再启动一个浏览器，通过浏览器来访问本机的服务。在 Jupyter Notebook 中，可以从服务器上传/下载文件，并编写自己的 `ipynb` 文件代码。

关于 Jupyter Notebook 工具的具体使用方法，这里不做过多介绍。在网上可以查到好多使用教程。

2.4 实例 1：运行 Python 程序，并传入参数

下面通过例子来介绍如何运行 Python 程序。

实例描述

创建一个扩展名为 `py` 的 Python 源代码文件。编写代码，将传入 Python 程序中的参数显示出来，并使用下列两种方式启动 Python 程序：

- (1) 在命令行中，启动 Python 程序并传入参数；
- (2) 在 Spyder 中，启动 Python 程序并传入参数。

2.4.1 在 Spyder 中新建 Python 文件，编写代码，运行代码

1. 新建文件

单击“新建文件”按钮，创建一个文件，如图 2-15 所示。

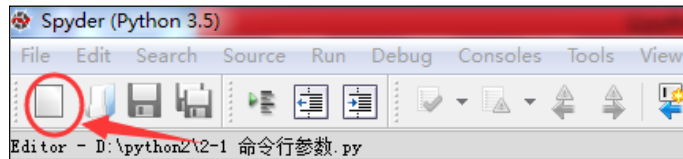


图 2-15 新建文件

2. 编写代码

单击“新建文件”按钮，会出现如图 2-9 所示的界面，在其中编写如下代码。

代码 2-1：命令行参数

```
import sys                                #导入 sys 模块
print ('参数个数为:', len(sys.argv), '个参数。')    #输出参数的个数
print ('参数列表:', str(sys.argv))                #输出参数的内容
```

第 1 行的代码是，引入了 `sys` 模块。在程序执行时，系统将启动参数传递给 `sys` 模块下的 `argv` 变量。

第 2 行的代码是，使用 `len` 函数来计算启动参数 `sys.argv` 的长度，并通过 `print` 函数将其输出到屏幕上。

第 3 行的代码是，使用 `str` 函数将启动参数 `sys.argv` 转化为字符串，并输出到屏幕上。

3. 运行程序

代码编写好之后，直接就可以运行了。

(1) 单击图 2-16 中箭头所指的按钮。

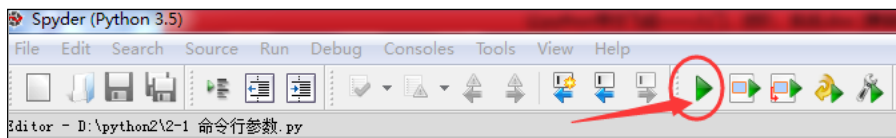


图 2-16 启动按钮

(2) 系统会提示是否要保存文件，这里将代码文件保存到本地硬盘（本例中保存的文件及路径为：d:/python2/2-1 命令行参数.py）。

(3) 保存结束后，程序便开始运行。将在输出栏输出结果，如图 2-17 所示。

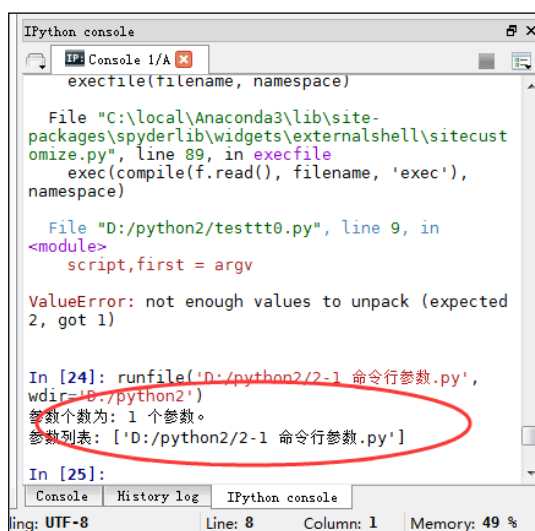


图 2-17 输出结果

图 2-17 中输出的内容为程序运行的结果。可以看到，默认的 Python 程序是有一个参数的。该参数的内容就是运行文件本身，即“D:/python2/2-1 命令行参数.py”（这正是作者演示的代码文件）。

2.4.2 用命令行启动 Python 程序，并传入参数

编写好代码之后，就可以传入参数将其启动了。

先来演示用命令行运行 Python 程序的方法：

(1) 单击“开始”按钮，输入“cmd”后按 Enter 键，如图 2-18 所示。

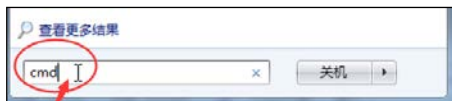


图 2-18 启动命令行

(2) 屏幕会弹出一个黑框的窗口，在窗口中输入如下命令：

```
C:\Users\Administrator>D:
D:\>cd python2
D:\python2> python "2-1 命令行参数.py" arg1 arg2
```

前两行的意思是来到代码文件所在的目录，最后一行是使用 python 命令启动程序文件，并传入两个参数 arg1、arg2。

(3) 按 Enter 键后显示如下结果：

```
参数个数为：3 个参数。
参数列表：['2-1 命令行参数.py', 'arg1', 'arg2']
```

这是程序的输出结果：第一行为参数的个数，第二行为参数的内容。

2.4.3 用 Spyder 启动 Python 程序，并传入参数

下面演示在 Spyder 中运行 Python 程序：

(1) 在图 2-12 中，单击标注为“2”的按钮。

(2) 弹出如图 2-19 所示对话框，在其中勾选图 2-19 所示复选框并填入参数，单击“OK”按钮。

(3) 单击图 2-16 中箭头所指的按钮，启动程序。

这时在输出窗口就会看到参数结果的显示：

```
参数个数为：3 个参数。
参数列表：['D:/python2/2-1 命令行参数.py', 'arg1', 'arg2']
```

通过这个例子，读者能够掌握 Spyder 的基本使用方法。当需要传入参数时，按照本案例的方法配置一下参数即可。

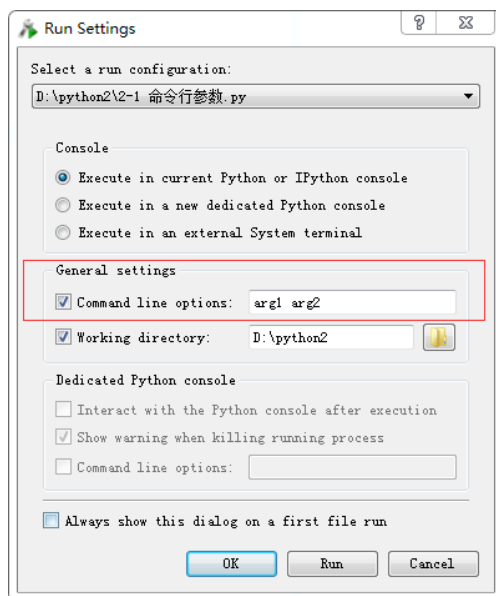


图 2-19 输入参数

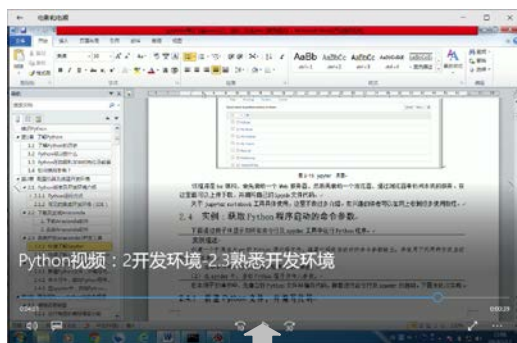
第 3 章

语言规则——Python的条条框框

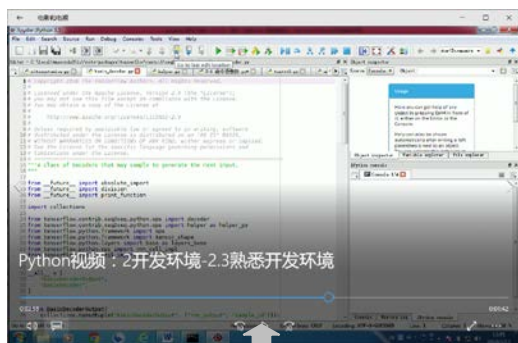
Python 属于编程语言的一种。在学习 Python 语言规则之前，有必要了解一下编程语言的各种分类，以及 Python 在其中所属的类型。了解这些之后，再来学习 Python 的基础语言规则和代码文件结构就会更容易理解。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解



具体代码操作演示



Python视频：
2开发环境
-2.1-2.2版本及
安装.mp4



Python视频：
2开发环境-2.3
熟悉开发环境.
mp4



Python视频：
2开发环境-2.4
实例1演示程
序.mp4

本章共有 4 段教学视频，总时长为 28 min 左右。包含以下内容：

- 讲述了 Python 语言类型及基础规则。
- 介绍了 Python 中模块与包的知识，还有模块的作用、分类、搜索路径、属性等内容。
- 讲解模块的四种导入方式。
- 通过 3.6 节的实例来演示模块的使用。

3.1 了解编程语言的分类

计算机发展到今天，编程语言已经是五花八门。在这么多语言中，Python 处于什么位置呢？为了更形象地描述 Python 语言特性，下面从多个角度来比较 Python 与其他语言的区别。

业界关于编程语言的分类标准有很多，有的从运行角度，有的则从形态角度。

1. 从运行角度的分类

从运行角度来看，编程语言的类型可以分为两种：编译型和解释型。

Python 属于解释型语言。

（1）解释型语言：代码可以直接运行。当然，这也是依赖于附加程序（解释器）来实现的。解释器会不断地循环一组动作：取一行源程序，将其转成二进制指令，然后执行。一直下去，直到全部的源程序读取完毕。类似这样的语言还有 JavaScript、VBScript、Perl 等。

（2）编译型语言：代码本身不能运行，需要一个附加程序（编译器）将其转换成由二进制代码组成的可执行文件，然后才可以运行。例如：C/C++、VB 等。



提示：

解释型语言和编译型语言最大的区别是：

- （1）解释型语言的代码必须依赖于解释器才可以执行；
- （2）编译型语言的代码生成可执行程序后，可以独立执行。

也有一些附属的打包程序，可以将解释型语言的代码打包成为独立执行的程序。但总体来讲，使用打包程序的效率及性能会略慢于编译型语言生成的程序。

2. 从形态角度的分类

从形态角度来看，编程语言的类型可以分为两种：动态语言和静态语言。

Python 属于动态语言。

(1) 动态语言：是指程序运行时可以改变其结构，可以对变量或函数进行修改。因为程序中的代码是在运行时才开始检查数据类型的，所以没有运行的语句是被程序忽略的。即，定义变量时不需要指定数据类型，只有在第一次给变量赋值时，根据赋值的类型在内部指定该变量的类型。类似的语言还有 Perl、Ruby 等。

(2) 静态语言：常用于编译型语言，在编译时需检查数据的类型。即，在使用变量之前必须要定义好数据类型。如：C、C++、C#、Java 等。

3. 语义角度的编程语言分类

从语义角度来看，编程语言的类型可以分为两种：强定义类型和弱定义类型。

Python 属于强定义类型。

(1) 强定义类型语言：会严格区分内部的变量类型。一旦指定了变量的类型，就必须经过转换才能存取为其他类型。类似的语言有 C、Java 等。

(2) 弱定义类型语言：是指不严格区分内部的变量类型，一般是只要大小放得下即可转化。类似的语言有汇编语言、VBScript、JavaScript 等。



提示：

Python 是一种动态解释型的强定义类型语言。

优点是：代码简洁、可读性强、开发效率比较高、可移植性好、可拓展性好、可嵌入性好；

缺点是：运行时的性能相对较低、源代码不能加密。

3.2 基础规则

对于一个 Python 项目，它的代码部署路径与代码内容也是息息相关的，这里面涉及代码文件、模块、包的相关概念及使用规则。下面分别介绍。

3.2.1 了解 Python 源代码相关的几个概念

使用 Python 语言编写的源代码，从粗到细可以分为模块文件、代码块、代码三部分。代码里又有“函数”和“类”两种概念。

(1) 模块文件：是指封装好的代码文件，可作为独立的模块被其他程序引用。

(2) 代码块：可以理解成为一个容器，其中可放置一条一条语句。一个模块文件、一个函数体、一个类、交互式命令中的单行代码，都可以称作一个代码块。

(3) 代码中的函数：在一个模块文件中，将若干条语句封装在一个代码块里以完成某个独立的功能，供其他程序使用。代码块是通过模块文件来承载的。

(4) 代码中的类：以面向对象思想，将变量及函数封装成具有某个类别特性的结构体。它可以模块文件的形式单独提供，也可以与本地程序一起放在相同代码文件中。

3.2.2 语句的基本规则：变量、语句、代码块

Python 语句的运行是在 Python 解释器中实现的。错误的 Python 语句不能被解释器所执行。

具体的语句规则如下：

1. 变量的命名

可以使用字符数字和下画线的组合，首字母可以是字母或者下划线。例如下列的名字都是合法的：

```
Code、_data、_name_、tt
```

2. 语句区分

在 Python 解释器中，源代码是一行一行被解释执行的，行与行之间通过换行符号来区分。默认是一行一条语句，如果在一行中放有多条语句可以通过“；”来区分。例如：

```
A=6           #一行就是一条语句
B=8;cc=10     #这种属于两条语句，虽然在一行，但是使用分号分开了
```

3. 代码块

通过缩进来表述“代码块”，即同一个代码块中的语句具有相同的缩进格式。缩进可以是

使用空格或是 Tab 键来实现。

代码块的概念有点抽象，可以通过下面的例子来理解：

```
def fun():           #定义一个函数名字叫作 fun
    print("hello fun") #该语句被缩进，表示 fun 函数的函数体，只有执行函数 fun 时才会被执行
print("hello main")  #该语句没被缩进，直接执行
fun()                #该语句没被缩进，会执行其函数体内的语句
```

上面的代码是一个函数的定义及调用。

第一行代码为函数定义，第二行代码为函数 `fun` 的代码块内容。程序运行时，会先执行第三行代码，输出“hello main”，接着执行第四行代码，调用函数 `fun`，输出“hello fun”。

3.2.3 添加注释

英文字符“#”用来代表注释。它的生效范围是“行”，即在一行代码中#之后的内容将不被 Python 解释器处理。

注释一般用来说明代码的具体含义，以方便开发人员阅读。另外，也可以用来控制程序暂时不执行某一部分代码块。例如：

```
#print("hello main")    #该语句前面有#号，表示为注释语句，不会被执行
```

上面的代码就不会被执行。



提示：

在 Spyder 中，还可以使用快捷操作快速批量添加或删除注释，具体见“2.3.1 快速了解 Spyder”小节中的“2. 注释功能”。

另外，也可以使用三个单引号（`'''`）或者三个双引号（`"""`）将代码段变为字符串，程序同样不会执行。例如：

```
'''                    #开始
def fun():
    print("hello fun")
print("hello main")
fun()
'''                    #结束
```

上面的代码被一对三个单引号所组成的语句包围，系统会认为这是一个字符串，不会将其当作语句来执行。

3.2.4 使用 Python 的“帮助”

在 Python 中使用 `help` 命令来获取帮助信息。它可以查找关于 Python 的基础函数、类型、常用库等信息。例如：

```
help(print)           #显示 print 函数的帮助信息
```

输出如下信息：

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

输出的信息中，描述了 `print` 函数的说明、定义、参数的意义。开发者拿到这个信息，就可以按照参数的说明来使用该函数了。

3.3 代码文件的结构

Python 的文件结构可以分为代码、模块、包。

代码是组成程序的基本单位，是一条一条的具体语句。代码的载体是文件。Python 中的代码文件是以 `.py` 结尾的。

下面分别介绍模块和包。

3.3.1 模块

在实现一个复杂程序时，需要编写的代码量往往很庞大。将所有的代码都放在一个文件里，显然很不合适。这时就需要将代码分割成多个文件，每个文件中放置功能相对独立的代码。在使用时，同时将多个文件中的代码导入到当前代码文件中，以实现完整的功能。这种被裁分后的仍具有独立功能的代码文件就是模块（Module）。

概括地说，模块是一个支持导入功能的，以.py 结尾的代码文件，如图 3-1 所示。

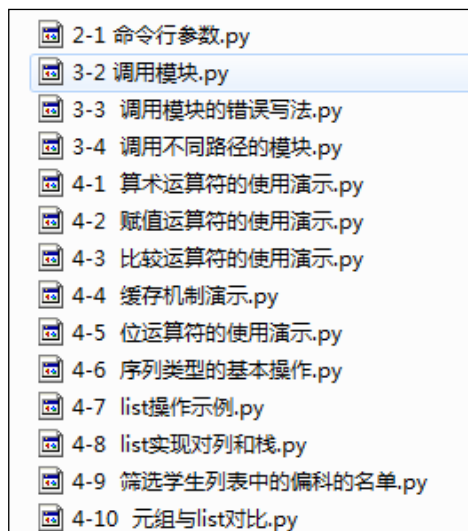


图 3-1 Python 文件

3.3.2 包

当模块足够庞大时，维护起来不太方便。这就需要使用包（Package）。包可以把多个模块（.py 文件）放在同一个文件夹中，以便归类与管理。

概括地说，包就是放置模块的文件夹。例如，下面代码是一个名字为 CycleGAN 对抗神经网络的目录：

```
CycleGAN/                                #顶级包
  __init__.py
  main.py
  sampel/                                #样本处理子包
    __init__.py
    Download_data.py
    build_data.py

  model/                                #模型子包
    __init__.py
    discriminator.py
    generator.py

  work/                                  #协调工作子包
```



```
__init__.py  
train.py  
export_graph.py  
eval.py
```

在上面的目录结构中，顶级包 `CycleGAN` 下面分为样本处理子包、模型子包和协调工作子包。每个子包里放有功能相近的模块。比如，模型子包里面放的模块都是关于各种模型的代码。

- `discriminator.py`: 用于实现判别器的模型；
- `generator.py`: 用于实现生成器的模型。



注意：

在每个包文件夹里都必须包含一个 `__init__.py` 文件。该文件的作用是，告诉 Python 环境该文件夹是一个包。`__init__.py` 可以是一个空文件。

3.4 模块的详细介绍

Python 模块（Module）包含了 Python 对象定义和 Python 语句。在模块里可以定义函数、类和变量。模块也能包含可执行的代码。

把相关的代码分配到一个模块里，可以使代码更好用、更易懂。养成使用模块的习惯，开发者能更有逻辑地组织自己的 Python 代码段。

3.4.1 模块的作用及分类

模块的主要作用是，用来被导入到其他代码模块中。将其他模块导入（`import`）到当前模块，可以理解成是对当前模块的一种功能增强。

不同模块中的函数名和变量是可以相同的。模块的使用，避免了庞大代码量中函数名和变量名冲突的问题。另外，将代码模块化也提高了代码的可维护性与重用性。

Python 中的模块可以分为内置模块、自定义模块和第三方模块三类。

- 内置模块：Python 中本来就有的模块；
- 自定义模块：自己开发的模块；
- 第三方模块：需要单独下载、安装并导入的模块。

3.4.2 模块的基本使用方法

模块使用起来非常简单，使用“import 模块名字”（中间有空格）即可将模块导入。

下面引入一个内置的 `time` 模块，以显示当前时间：

```
import time                                #引入 time 模块
print(time.time())                        #获取当前时间，输出：1513299326.049188
print( time.localtime( time.time() ) .tm_year)  #将当前时间格式化，输出：2018
```

对于内置模块和已经安装好的第三方模块，系统会自动在内置或已经配置好的模块路径下查找该模块，并载入。

3.4.3 模块的搜索路径

自定义模块，还需考虑模块所在的位置，要结合“包名”一起引入。具体语法是：

```
import 包名.模块名
```

例如，在 3.3.2 小节的例子中，想要在 `main.py` 中导入 `work` 下面的 `train.py` 模块，需要如下写法：

```
import work.train
```

当导入名为 `train` 的模块时，解释器会先尝试从内置模块匹配；如果没找到，则将在 `sys.path` 记录的所有目录中搜索 `train.py` 文件。变量 `sys.path` 是一个字符串列表，它为解释器指定了模块的搜索路径。`sys.path` 包括：

- 当前程序所在目录；
- 标准库的安装目录（例如：`pythom35\lib\site-packages`）；
- 操作系统环境变量 `PythonPATH` 所包含的目录。

在编写代码时，也可以通过列表操作来对 `sys.path` 进行读写，例如：

```
import sys                                #引入 sys 库
print(sys.path)                          #将 sys.path 打印出来
sys.path.append('d:// lib//Python')      #在 sys.path 里添加一条路径
```

通过代码对 `sys.path` 进行修改，减少了 Python 程序对环境的依赖，为部署提供了便利。

对 `sys.path` 的修改，只在本次程序内有效，系统并不会将 `sys.path` 永久保存。如想永久生效，还需在环境变量里进行配置。

3.4.4 模块的属性

模块除了被引用以外，还会有自己的属性可供调用者查看。其属性大致有如下几种：

- `__name__`：名字；
- `__doc__`：详细说明，介绍了该模块的使用方法；
- `__package__`：所在的包名；
- `__loader__`：加载的类名；
- `__spec__`：简介，介绍了该模块的名字、加载类名、来源类型等概要信息。

要想查看这些属性的内容，可以通过“导入的模块名 + . + 具体的属性变量”，例如：

```
import time                #引入time 模块
print(time.__name__)      #模块名字。输出: time
print(time.__doc__)       #详细说明。输出: This module provides various.....
print(time.__package__)   #包名。因为是内置模块，包名为空，所以输出为空
print(time.__loader__)    #加载的类名。输出: <class '_frozen_importlib.BuiltinImporter'>
print(time.__spec__)      #简介。输出: ModuleSpec(name='time', .....
```

3.4.5 模块名字的可变性

在 Python 中，模块的名字属性会根据不同的使用场景发生变化。当模块被导入到其他模块时，`__name__` 的值为模块本身的名字；而当该模块自己独立运行时，`__name__` 的值会变为“`__main__`”。例如：

(1) 新建一个 `test1.py` 文件，写入如下代码：

```
print(__name__)           #将当前模块的名称打印出来
```

(2) 新建一个 `test2.py` 文件，写入如下代码：

```
import test1              #导入 test1 模块
```

(3) 运行 `test1.py` 文件，输出如下：

```
__main__                  #输出模块名
```

(4) 运行 `test2.py` 文件，输出如下：

```
test1                     #输出模块名
```

可以看到，虽然是同样一段代码，但在第 (3) 步中是直接运行的，输出了模块名字为“`__main__`”，而在第 (4) 步是通过 `import` 导入运行的，输出的模块名字就变为了“`test1`”。

3.4.6 模块的常规写法

Python 程序中，每一个代码文件在可以独立运行的同时，也可以作为一个模块文件。编写模块文件的好习惯是，在编写该模块提供的函数或类的同时，也可以把自身当作独立运行的文件来为自身模块做单元测试。这就需要借助模块名字属性的可变特性来实现了。

例如，可以在模块的最下面加入名字的判断，并执行单元测试代码：

```
if __name__ == '__main__':  
    执行单元测试代码
```

这样，当直接运行这个模块文件时，可以通过测试代码来检验所定义的函数的输入和输出是否正确。而引入模块时，测试代码不会被执行。为模块编写对应的单元测试代码，是一个非常好的编程习惯。

3.5 模块的四种导入方式

根据所要引入模块的内容不同，可以将模块的导入方式分为四种。下面依次介绍。

3.5.1 import as 方式

import as 的方式其实是实现了两步操作：先将模块导入，再为模块重命名。其写法如下：

```
import a as b
```

其中，a 代表要引入的模块，b 是将 a 重命名后的名称。即，将模块 a 导入，并将其重命名为 b。

举例如下：

```
import time as x          #导入模块 time 并将其重命名为 x  
s = x.ctime()             #调用模块 time 中的 ctime 函数，得到当前时间  
print(s)                  #将时间输出：Thu Mar 1 14:59:17 2018
```

例子中，把引入的 time 模块重命名为 x，后面就可以使用 x 来调用 time 模块，并调用其 ctime 方法以获得当前时间。

3.5.2 from import 方式

from import 方式是直接把模块内的函数或变量的名称导入当前模块。其写法如下：

```
from a import func
```

上述代码表示，将模块 `a` 中的 `func` 函数导入到当前模块。使用这种方式导入时，当前模块将只能使用 `a` 中的 `func` 函数，无法使用 `a` 中的其他函数。

例如：

```
from time import ctime    #导入模块 time 中的 ctime 函数
s = ctime()               #直接调用函数
print(s)                  #将时间输出: Thu Mar  1 14:59:17 2018
```

例子中，先从 `time` 模块中导入 `ctime` 函数，接着就可以直接使用 `ctime` 函数来获得时间。因为只导入了 `time` 模块中的一个函数，如想再调用 `time` 模块中的其他函数，则系统会报错。

例如，接上面代码，添加如下语句：

```
print( time.time())
```

运行时，程序会报错：

```
NameError: name 'time' is not defined
```

因为程序只引入了 `ctime` 函数，并不知道 `time` 模块，所以提示 `time` 没定义。

3.5.3 from import * 方式

`from import *` 方式将模块中所有的名字（以下画线开头的名字除外）导入到当前模块符号表里。

具体代码如下：

```
from time import *        #导入模块 time 中的所有名字（以下画线开头除外）
s = ctime()               #直接调用函数
print(s)                  #将时间输出: Thu Mar  1 14:59:17 2018
```

这时再运行 `time` 中的 `time` 函数就不会报错了。代码如下：

```
print( time())            #返回当前时间的时间戳(1970 纪元后经过的浮点秒数)输出:1519889455.5011432
```



提示：

使用 `from import*` 方式导入模块的好处是方便，弊端是容易产生名字冲突（如果当前模块里也有个 `ctime` 函数，就会跟导入的 `ctime` 发生名字冲突）。这是需要注意的地方。

3.5.4 导入文件名中带空格的模块

在 3.3.1 小节中介绍过，模块是一个代码文件。而代码文件的命名规范是符合操作系统的命名规则的。在操作系统中允许文件名中出现空格，但这个规则与 Python 语法发生冲突，因为在 Python 中是以空格来隔离一行语句中的不同元素的。在这种情况下，导入带空格的模型会带来麻烦。

解决办法是使用 `__import__` 方法。假设有个模块叫作“9-24 yuyinutils”，要在当前文件引入它时，可以写成如下样子：

```
yuyinutils = __import__("9-24 yuyinutils")
```

使用 `__import__` 方法，将模块名称用字符串的方式传入，就可以得到该模块的引用了。



注意：

该部分代码来自于《深度学习之 TensorFlow：入门、原理与进阶实战》一书中语音识别的“案例 9-23 yuyinchall.py”代码文件中的开头部分。如何将 `__import__` 的返回值当作模块使用，可以参考这部分代码。

除了模块名字中间带空格的情况外，如模块名字是以数字开头的，也无法正常导入，需要使用 `__import__` 方法才可以解决这个问题。

3.6 实例 2：封装获取系统信息的模块，并将其导入

下面通过实例演示导入自定义模块的具体操作。

实例描述

创建一个扩展名为 `py` 的 Python 源代码文件，并将其作为模块。该模块的功能是输出当前系统信息。按照下列两种情况将模块导入并使用：

- (1) 在当前代码中的函数与模块中的函数同名情况下，导入模块。
 - (2) 在模块与当前代码不在同一路径的情况下，导入模块
-

例子中举出的几种情况，都是实际开发中常会出现的情况，也是最容易出问题的情况。下面来依次实现。

3.6.1 在当前代码中的函数与模块中的函数同名情况下，导入模块

先创建模块，实现系统信息的输出，再编写代码调用模块。

1. 创建模块

首先需要创建一个代码文件用作模块，并命名为“getenv.py”。在模块中实现一个函数，令其打印系统信息。代码如下：

代码 3-1: getenv

```
import platform
import sys
import os

def showENV():                                #函数
    s = platform.platform()
    print("当前系统: ",s)                    #获取系统信息
    p = sys.path
    print("当前安装路径: ",p)                #获取安装路径
    op = os.getcwd()
    print("当前代码路径: ",op)               #获取当前代码路径
    print("Python 版本信息: ",sys.version_info)

if __name__ == '__main__':                    #进行模块的单元测试
    showENV()
```

上面代码中实现了一个函数 showENV，在函数 showENV 中输出了关于系统环境的相关信息。最后两行代码为该模块的测试代码，使得该模块可以作为一个独立运行的程序。代码运行后得到如下输出：

```
当前系统: Windows-7-6.1.7601-SP1
当前安装路径: ['', 'C:\\local\\Anaconda3\\lib\\site-packages\\pymysql-0.7.11-py3.5.egg', 'C:\\local\\Anaconda3\\python35.zip', 'C:\\local\\Anaconda3\\DLLs', 'C:\\local\\Anaconda3\\lib', 'C:\\local\\Anaconda3', 'c:\\local\\anaconda3\\lib\\site-packages\\setuptools-23.0.0-py3.5.egg', 'C:\\local\\Anaconda3\\lib\\site-packages', 'C:\\local\\Anaconda3\\lib\\site-packages\\Sphinx-1.4.1-py3.5.egg', 'C:\\local\\Anaconda3\\lib\\site-packages\\win32', 'C:\\local\\Anaconda3\\lib\\site-packages\\win32\\lib', 'C:\\local\\Anaconda3\\lib\\site-packages\\Pythonwin', 'C:\\local\\Anaconda3\\lib\\site-packages\\IPython\\extensions', 'C:\\Users\\Administrator\\.ipython']
当前代码路径: D:\\python2
Python 版本信息: sys.version_info(major=3, minor=5, micro=2, releaselevel='final', serial=0)
```

从上面的输出中可以看出，输出的环境信息有：当前操作系统的版本、安装路径、代码路径、版本信息。函数 `showENV` 的相关知识，读者可以先不用去关心。第 6 章还会有详细的介绍。

2. 编写信息调用模块

创建另一个代码文件“3-2 调用模块”，并在其中实现自己的 `showENV` 函数。代码如下：

代码 3-2：调用模块

| | |
|---|--------------------------------|
| <code>import getenv</code> | <code>#导入自定义模块 getenv</code> |
| <code>def showENV():</code> <code> print("this is my env")</code> | <code>#实现同名函数 showENV</code> |
| <code>showENV()</code> | <code>#调用本地函数</code> |
| <code>getenv.showENV()</code> | <code>#调用 getenv 模块里的函数</code> |

在代码的最后两行，进行了本地函数和自定义模块里的函数调用。代码运行后输出如下：

```
this is my env
当前系统: Windows-7-6.1.7601-SP1
.....
```

输出结果的第一行为本地 `showENV` 的输出，第二行为 `getenv` 模块里的 `showENV` 输出。

3. 调用模块的错误写法——模块函数被覆盖

如果用 3.5.3 小节的方式直接用 `import *` 导入全部符号（见代码 3-3），将导致函数名被覆盖。

代码 3-3：调用模块的错误写法

| | |
|---|-------------------------------|
| <code>01 from getenv import *</code> | <code>#导入 getenv 的全部符号</code> |
| <code>02</code> | |
| <code>03 def showENV():</code> | |
| <code>04 print("this is my env")</code> | |
| <code>05</code> | |
| <code>06 showENV()</code> | <code>#调用 showENV 函数</code> |

代码运行后，输出如下结果：

```
this is my env
```

从结果看出，没有输出系统信息。这说明模块中的函数被覆盖。

4. 调用模块错误写法——本地函数被覆盖

如果将导入模块的代码移到 showENV 的下面（见代码 3-3），将会导致当前函数被覆盖。

代码 3-3：调用模块的错误写法（续）

```
07 def showENV():
08     print("this is my env")
09 from getenv import *           #导入 getenv 的全部符号
10
11 showENV()                     #调用 showENV 函数
```

代码运行后，将会输出系统信息，如下：

```
当前系统: Windows-7-6.1.7601-SP1
.....
```

这次没有显示出当前函数所打印的语句，是因为模块中的函数覆盖了当前函数。



注意：

本小节的 3 和 4 小标题的错误写法都是由于使用了 import * 的方法，并且模块函数与本地函数重名的情况造成的。编写代码时需额外注意。

通过这两种错误的写法也可以发现，在 Python 中的符号生效规则是“就近”。即调用函数时，系统会找离该函数最近的代码定义来执行。

3.6.2 在模块与当前代码不在同一路径的情况下，导入模块

下面做个实验。在当前代码的同级目录下，新建一个文件夹“model3”。将写好的自定义模块“getenv.py”移到当前代码的下一级目录“model3”里。编写代码将自定义模块“getenv.py”载入，并调用其内部函数，见代码 3-4。

代码 3-4：调用不同路径的模块

```
import model3.getenv as getenv           #导入自定义模块 getenv

def showENV():                           #实现同名函数 showENV
    print("this is my env")

showENV()                                #调用本地函数
getenv.showENV()                         #调用 getenv 模块里的函数
```

在导入模块 `getenv` 时，需要指定 `getenv` 的上一级文件夹名称（见代码的第 1 行）。代码运行后，输出了正确的结果，如下：

```
this is my env
当前系统: Windows-7-6.1.7601-SP1
.....
```

3.6.3 导入上级路径的模块

如果 `getenv.py` 文件想要导入上一级（与 `model3` 同级）的模块，可以使用“`import ..模块名`”的方式。这里就不再举例。读者可以自行尝试。

第 2 篇 进阶

本篇介绍了 Python 编程的基础知识，其中包括变量的介绍与操作、控制流、函数操作、错误与异常、文件操作相关的知识。每一章都有若干案例，通过这一系列的学习，可以帮助读者全面掌握 Python 的编程基础。有了扎实的基础之后，在后面的高级 Python 应用部分才不会感觉到吃力。

- ▶ 第 4 章 变量与操作——编写代码的基石
- ▶ 第 5 章 控制流——控制执行顺序的开关
- ▶ 第 6 章 函数操作——功能化程序片段的封装
- ▶ 第 7 章 错误与异常——调教出听话的程序
- ▶ 第 8 章 文件操作——数据持久化的一种手段

第 4 章

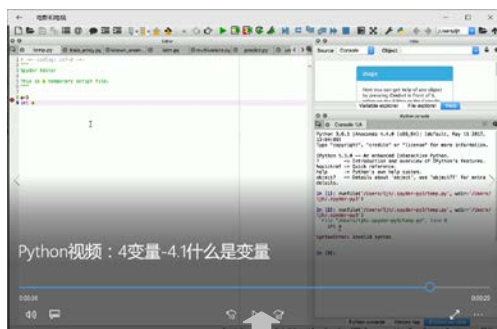
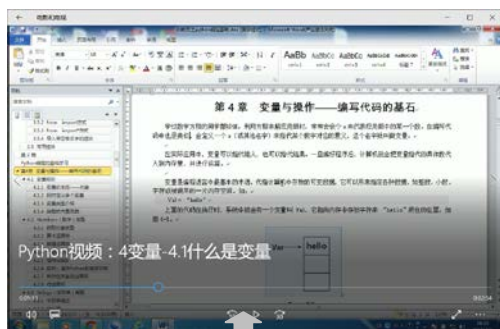
变量——编写代码的基石

学过数学方程的读者都知道：利用方程来解应用题时，常常会设置一个 x 来代表应用题中的某一个数。在编写代码中也类似，会定义一个 x （或其他名字）来指代某个数字对应的意义，这个 x 就是作变量。

在实际应用中，变量可以指代输入，也可以指代结果。一旦编好程序后，计算机就会把变量指代的具体数代入到内存里，并进行运算。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解

具体代码操作演示



Python视频：4
变量-4.1什么是
变量.mp4



Python视频：4
变量-4.2了解变
量的规则.mp4



Python视频：4
变量-4.3numbers类
型.mp4



Python视频：4
变量-4.4string类
型.mp4



Python视频：4
变量-4.5列表类
型-实例100列和
栈.mp4



Python视频：4
变量-4.5列表类
型-实例100列和
栈.mp4



Python视频：4
变量-4.6元组类
型.mp4



Python视频：4
变量-4.7集合类型
.mp4



Python视频：4
变量-4.8字典类
型.mp4



Python视频：4
变量-4.9序列排
列.mp4

本章共有 10 段教学视频，总时长为 102 min 左右。包含以下内容：

- 讲述了什么是变量。
- 介绍了 Python 中变量的规则。
- 讲解 numbers 类型。
- 讲解 string 类型。
- 讲解列表类型，并通过实例 9 演示了列表的基本操作。
- 讲解列表类型，并演示了使用列表实现队列和栈的实例。
- 讲解元组类型。
- 讲解集合类型。
- 讲解字典类型。
- 讲解“深、浅拷贝”的知识。

4.1 什么是变量

变量是编程语言中最基本的术语，用来代指计算机中存放的可变数据——如整数、小数、字符或被调用的一片内存空间。如：

```
Val = "hello"
```

上面的代码在执行时，系统中就会有有一个变量叫 Val，它指向内存中存放字符串“hello”所在的位置，如图 4-1 所示。

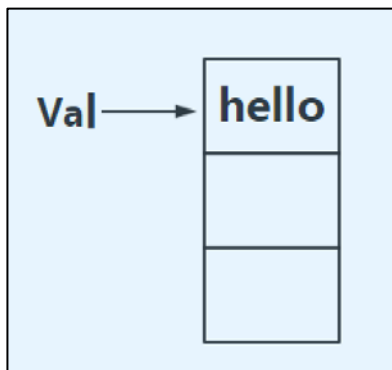


图 4-1 变量

定义了变量之后，可以通过编写具体的逻辑对变量进行操作计算，从而实现想要的功能。

4.2 了解变量的规则

在实际情况中，程序需要处理不同类型的变量，所以在 Python 内部，变量是要分为各种类型的。比如，整型变量只可以代表整数，浮点型变量只可以代表带小数点的数字，字符型变量只可以代表字符。

编写 Python 代码过程中，定义变量与变量赋值必须在同一步完成。Python 内部会根据所赋变量的类型来创建该变量。

例如：定义整型变量 `a` 等于 5，可以写成 `a=5`。这时 Python 会在内存中创建一个类型为整型的变量 `a`，并让这个 `a` 的值为 5。在后面的代码中用到 `a` 就相当于用到了数值 5。

4.2.1 明白变量的本质——对象

Python 语言可以根据赋值的语句来自动创建对应类型的变量，它是怎么实现的呢？

Python 内部使用了一个对象模型，这个对象模型用来储存变量及其对应的数据。在 Python 语言中，任何类型的变量都会被翻译成一个对象，这就是变量的本质。

Python 内部的对象模型由 3 部分组成：身份、类型和值，具体意义如下。

- 身份：用来标识对象的唯一的标识。通过调用函数 `id` 可以得到它。这个身份标识值可以被理解成该对象的内存地址。
- 类型：用来表明对象可以存放的类型。具体的类型限制了该对象保存的内容、可以进行的操作、遵循的规则。想要查看某个对象的类型可以调用函数 `type`。
- 值：对象所存储的具体数值。

在 4.2.2 小节中会演示 `id` 函数与 `type` 函数的使用。

4.2.2 同时定义多个变量

实际写代码中，可以通过一条语句定义多个变量，彼此之间用逗号隔开即可。

例如，在一条语句中为多个变量同时赋值，如下：

```
var1,var2,var3=value1,value2,value3
```

这时，系统会根据 `value1`、`value2`、`value3` 的值类型，分别定义 `var1`、`var2`、`var3` 三个变量，并为它们赋值。例如：

```
var1,var2,var3 = 2,5,3                                #定义 var1 为 2, var2 为 5, var3 为 3
print("指针:",id(var1),"类型: ",type(var1),"值: ",var1)    #输出变量 var1、var2、var3
的指针、类型、值
print("指针:",id(var2),"类型: ",type(var2),"值: ",var2)
print("指针:",id(var3),"类型: ",type(var3),"值: ",var3)
```

这段程序运行后，输出的结果为：

```
指针: 1650393616 类型: <class 'int'> 值: 2
指针: 1650393712 类型: <class 'int'> 值: 5
指针: 1650393648 类型: <class 'int'> 值: 3
```

4.2.3 变量类型介绍

不同的 Python 版本对类型的归类略有差异，这里以 Python 3 为主。在 Python 3 中，所划分的类型有六种：

- numbers：数字。
- string：字符串。
- tuple：元组。
- sets：集合。
- dictionaries：字典。

其中，numbers 类型是一个数值类型的合集，具体又可以细分为 int（整型）、float（浮点型）、bool（布尔型）、complex（复数）等类型。同时每个类型都有其自身的特点和规则，在后文中会一一介绍。

4.2.4 变量类型的帮助函数

Python 将自身所支持的变量类型定义成了一个个的类，任何类型（numbers、string、list……）都是一个类。每个类都有自己的方法和属性，通过类的方法和属性来规范该类型数值的操作。

Python 中内置了两个帮助函数。

- dir：用来查询类或者类型的所有属性，如 dir(list)。
- help：用来查询类或者类型具体的说明文档，如 help(int)。

在阅读下文之前，读者可以使用这两个函数对 Python 中的类型进行查询，看看系统是如何介绍具体类型的。

4.3 numbers（数字）类型

Python 3 中的 numbers 类型与大多数编程语言类似，它是一个数值类型的合集，具体又可以细分为 int（整型）、float（浮点型）、bool（布尔型）、complex（复数）等类型。

4.3.1 获取对象的类型

在 4.2.1 小节中介绍过，每个对象都可以通过内置的函数 type 来查询该变量所指的对象类型。这里使用如下代码来演示 type 函数的使用：

```
a, b, c, d = 32.6, 58, True, 8+7j ##定义四个变量 a、b、c、d，为它们赋予不同类型的值
print(type(a), type(b), type(c), type(d))    #将这四个变量的类型打印出来
```

运行后会得到如下结果：

```
<class 'float'> <class 'int'> <class 'bool'> <class 'complex'>
```

结果中的四个尖括号内的内容表示了对应变量的类型。即，a 的类型是 float，b 的类型是 int，c 的类型是 bool，d 的类型是 complex。

函数 type 对于所有的 Python 类型（包括后文中还会讲解的 string、list 等类型）都是有效的，其用法也是一样。

4.3.2 算术运算符

numbers 类型支持多种算术运算处理，具体的算术运算符见表 4-1。

表 4-1 算术运算符

| 运 算 符 | 描 述 |
|----------------|-----------------------------------|
| + | 加 |
| - | 减 |
| * | 乘 |
| / | 除，会生成浮点数结果 |
| % | 取模（余数） |
| ** 或 pow(x, y) | 幂。例如：3**4 等价与 pow(3,4)，即 3 的 4 次方 |
| // | 取整除 |
| abs(x) | 取绝对值 |

续表

| 运 算 符 | 描 述 |
|----------------|---|
| int(x,[base]) | 将 x 转换为整型。 x 可以是字符串或其他数字；base 是可选参数，默认为 10，表示将字符串 x 以 10 进制转化为整数。 当 base 被赋值时，x 必须是字符串；当 x 为浮点数时，转成的整数会将小数点后面全部舍掉。 如果想要更精确的转化，推荐用 math 库里面的 floor 和 ceil 函数来明确转换方式 |
| float(x) | 将 x 转换为浮点型 |
| complex(re,im) | 生成复数。re 为实数部分，im 为虚数部分。 例如：complex(8,7)，则生成一个复数 8+7j |
| c.conjugate() | 取 c 的共轭复数，假如 c=8+7j，则 c.conjugate() = 8-7j |
| divmod(x, y) | 返回商和余数。例如：divmod(13, 4) = (3, 1) |

大家在这里只需大致了解一下，下面将具体使用。

4.3.3 实例 3：演示“算术运算符”的使用

下面通过一段程序来演示“算术运算符”的使用。

实例描述

定义两个变量，分别按照表 4-1 中的运算符对其运算操作，观察运算结果。

定义两个变量 a 和 b，其中 a 的值为 20，b 的值为-3，按照表 4-1 中的运算符，对其进行加、减、乘、除等操作。代码如下：

代码 4-1：算术运算符的使用演示

```
a=20
b=-3
print(a+b)           #加号运算，输出：17
print(a-b)           #减号运算，输出：23
print(a*b)           #乘号运算，输出：-60
print(a/b)           #除号运算，输出：-6.666666666666667
print(a%b)           #取余运算，输出：-1
print(a**b)          #幂运算，输出：0.000125
print(a//b)          #整除运算，输出：-7
print(abs(b))        #绝对值运算，输出：3
print(int("1010",2)) #将字符串以二进制转换成整数，输出：10
print(float("3.14")) #将字符串换成浮点数，输出：3.14

c =complex(a,b)      #生成复数，输出：(20-3j)
print(c)
```

```
print(c.conjugate()) #计算共轭复数，输出：(20+3j)
```

```
print(divmod(a, b)) #计算除数与余数，输出：(-7, -1)
```

上面的代码实现了表 4-1 中列出的运算符的全部操作。在每行代码后面，都配有该代码输出的结果。读者可以看着代码参考输出结果，自行体会每个操作符的含义。

4.3.4 赋值运算符

下面来介绍 numbers 类型中所支持的赋值运算符，见表 4-2。

表 4-2 赋值运算符

| 运 算 符 | 描 述 |
|-------|-------------------------|
| += | 加。例如：a+=b 等价于 a=a+b |
| -= | 减。例如：a-=b 等价于 a=a-b |
| *= | 乘。例如：a*=b 等价于 a=a*b |
| /= | 除。例如：a/=b 等价于 a=a/b |
| %= | 取模。例如：a%=b 等价于 a=a%b |
| **= | 幂。例如：a**=b 等价于 a=a**b |
| //= | 取整除。例如：a//=b 等价于 a=a//b |

大家在这里只需大致了解一下，下面将具体使用。

4.3.5 实例 4：演示“赋值运算符”的使用

下面通过一段程序来演示“赋值运算符”的使用。

实例描述

定义两个变量，分别按照表 4-2 中的运算符对其运算操作，观察运算结果。

定义两个变量 a 和 b，其中 a 的值为 9，b 的值为 3，按照表 4-2 中的赋值运算符，对其进行加、减、乘、除等操作。代码如下：

代码 4-2： 赋值运算符的使用演示

```
a=9;b=3
a+=b          #加。等价于 a=a+b
print(a)      #输出：12
```

```
a=9;b=3
a-=b          #加。等价于 a=a-b
print(a)      #输出：6

a=9;b=3
a*=b          #加。等价于 a=a*b
print(a)      #输出：27

a=9;b=3
a/=b          #加。等价于 a=a/b
print(a)      #输出：3.0

a=9;b=3
a%=b          #加。等价于 a=a%b
print(a)      #输出：0

a=9;b=3
a**=b         #加。等价于 a=a**b
print(a)      #输出：729

a=9;b=3
a//=b         #加。等价于 a=a//b
print(a)      #输出：3
```

上面的代码实现了表 4-2 中列出的运算符的全部操作。在每行代码后面，都配有该代码输出的结果。读者可以看着代码参考输出结果，自行体会每个操作符的含义。

4.3.6 比较运算符

Python 中的比较运算符常常用来比较 numbers 类型数据。下面来介绍 numbers 类型中所支持的比较运算符，见表 4-3。

表 4-3 比较运算符

| 运 算 符 | 描 述 |
|----------|------|
| == | 等于 |
| != 或者 <> | 不等于 |
| > | 大于 |
| < | 小于 |
| >= | 大于等于 |
| <= | 小于等于 |

续表

| 运 算 符 | 描 述 |
|--------|-------|
| is | 指针等于 |
| is not | 指针不等于 |

大家在这里只需大致了解一下，下面将具体使用。



注意：

在 4.3 节讲解的字符串类型也可以使用比较运算符，但本质也是将字符串中字符的 ASCII 码数值拿出来比较。

自定义类型的比较方法，可以通过重载表 4-3 中的符号来进行实现（这部分内容见第 9 章）。

4.3.7 实例 5：演示“比较运算符”的使用

比较运算符有两种使用方法：一种是常规的方法——直接使用；另一种是链式使用。

下面通过一段程序来依次演示。

实例描述

- （1）定义三个变量，分别按照表 4-3 中的运算符对其运算操作，观察运算结果。
- （2）定义三个变量，使用链式比较符对其进行操作，观察运算结果。

1. “比较运算符”的常规使用方法

定义三个变量 a、b 和 c。其中 a 的值为 3，b 的值为 5，c 为 None。None 是 Python 中的一个特有的关键字，代表空值的意义，即没有任何值。

代码 4-3 按照表 4-3 中的比较运算符，对其进行大于、小于、等于、不等于等操作。

代码 4-3：“比较运算符”的使用演示

```
01 a=3
02 b=5
03 c=None
04
05 print(a==b)           #等于比较，输出：False
06 print(a!=b)           #不等于比较，输出：True
07 print(a>b)            #大于比较，输出：False
08 print(a<b)            #小于比较，输出：True
09 print(a>=b)           #大于等于比较，输出：False
```

```

10 print(a<=b)           #小于等于比较，输出：True
11 print(a is b)         #指针等于比较，输出：False
12 print(a is not b)     #指针不等于比较，输出：True
13 print(c is None)      #None 值等于比较，输出：True

```

上面的每行代码后面，都配有该代码输出的结果。读者可以看着代码参考输出结果，自行体会每个操作符的含义。

2. 链式使用“比较运算符”

链式表达式，就是在一行代码上出现两个比较运算符。接上面的代码，修改 `c` 的值为 7。使用链式表达式，进行 `a`、`b`、`c` 三个变量的比较。代码如下：

代码 4-3： 比较运算符的使用演示（续）

```

14 c=7
15 print(a<b<c)          #链式比较，输出：True

```

这两行代码的输出为 `True`。表明 `a<b<c` 的式子是成立的。在使用链式表达式时，需要考虑优先级的問題。

表 4-3 中列出的 8 个比较运算符，具有相同的优先级。在使用 `x < y < z` 这样的链式比较式时，它会从左到右的顺序来进行计算，相当于 `x < y and y <= z`。

3. “比较运算符”与“算术运算符”的优先级比较

“比较运算符”的优先级会比“算术运算符”低，例如下面的情况：

```
print(1.1==1.1-0.1)      #会输出 False
```

上面的代码会先计算 `1.1-0.1`，得到结果 1.0；然后再与 1.1 进行比较，得到结果 `False`。

尽管 Python 内部有这样优先级的规则，但还是建议大家在写类似这样的代码时为后面的算术运算加上括号。例如：

```
print(1.1==(1.1-0.1))    #将后面的算术运算括起来，表示优先计算
```

像上面这样的代码具有更好的可读性，代码易读，最大程度地消除代码歧义。这样写是一个优秀工程师的开发习惯。

4. 复数的比较

复数类型的对象不能比较大小，只能比较是否相等。

4.3.8 慎用 is 函数

表 4-3 中的倒数第二项介绍了 is 函数。is 函数的作用是比较指针是否相等。这里的指针代表存放对象的内存地址，即，is 函数是比较两个对象所在的内存是否相等。

那么 is 函数与 “==” 的区别是什么呢？

- “==” 只是判断两个对象的内容是否相等。
- is 函数不仅比较对内容是否相等，还比较指针是否相等。



提示：

两个对象如果使用 “==” 得到的是 true，使用 is 得到的不一定是 true，因为它们的地址有可能会不同；但是如果使用 is 得到的是 true，使用 “==” 就会一定是 true，因为它们本身就是一块内存的内容。

例如：

```
a=-256
b=-256
print(a==b)           #判断 a 和 b 的内容是否相等
print(a is b)         #判断 a 和 b 的内存是否相等
print(id(a),id(b))    #id 的意思是打印出对象的地址
```

执行代码，输出如下结果：

```
True
False
151744048 151743312
```

可以看出虽然 a 和 b 都是-256，但是它们的地址并不相等。

当然上面的例子只是一个很小的知识点。Python 在优化效率时做了好多工作，这导致存储对象的内存分配有很多隐含的规则。如不了解这些规则，使用 is 函数就会带来很多意想不到的麻烦。



提示：

能使用 “==” 处理的地方，尽量不要使用 is 函数。

Python 中关于对象内存的分配规则是什么样的呢？一起来往下学习。

1. 赋值的机制

在 Python 中，等号赋值是指直接将对象的内存指针赋值，如下面的例子：

```
x=23
y =45
print("x,y:",id(x),id(y),x,y)
x= y
print("x,y:",id(x),id(y),x,y)
```

执行代码，输出结果如下：

```
x,y: 1725105328 1725106032 23 45
x,y: 1725106032 1725106032 45 45
```

可以看到，x 与 y 在赋值之前各自有自己的指针，而当赋值之后，x 与 y 不仅有相同的值，而且还有相同的指针。

在赋值之前的 x=23 中，1725105328 所指的内存地址去哪了呢？其实这个 23 还在内存中，当系统判断没有变量引用该内存地址时，系统会使用内存回收机制，按设定好的规则回收该部分内存地址。

2. 缓存的重用机制

在优化整数对象时，Python 会根据对象的读取频繁程度以及内存占用情况来考虑，按照一定规则把某些对象存入缓存中。当程序的其他部分代码使用该值时，系统会先去缓存中查找该值，并直接引用找到的缓存地址，不需要额外创建。具体规则见表 4-4。

表 4-4 缓存规则

| 对 象 | 是否需要重新创新 | 生效范围 |
|--------------------|---------------------------|------|
| 范围在[-5, 256]之间的小整数 | 如之前在程序中创建过，就直接存入缓存，后续不再创建 | 全局 |
| 字符串对象 | | |
| 大于 256 的整数 | 只要在该代码块内创建过就直接缓存。后续不再创建 | 本代码块 |
| 大于 0 的浮点型对象 | | |
| 小于 0 的浮点型对象 | 不进行缓存，每次都需要额外创建 | |
| 小于-5 的整数 | | |

4.3.9 实例 6：演示 Python 的缓存机制

下面通过一段程序来演示 Python 缓存具体的规则。

实例描述

定义两个变量，为它们赋予相同的值，并将这两个变量的指针打印出来，然后分别按照缓存的具体规则分别为它们赋值，观察赋值后的两个变量的指针是否相等。如果相等，则表明在 Python 内部对其使用了缓存机制。

示例代码中使用了 6 种情况进行对缓存机制的演示，分别对应于 4.3.8 小节“2. 缓存的重用机制”中列出的具体规则。

代码 4-4： 缓存机制演示

```
01 #范围在[-5, 256]之间的小整数
02 int1=-5
03 int2=-5
04 print("[-5, 256]情况下的两个变量指针",id(int1),id(int2))
05
06 #对于字符串
07 s1="3344"
08 s2= "3344"
09 print("字符串情况下的两个变量指针",id(s1),id(s2))
10
11 #大于 256 的整数
12 int3=257
13 int4=257
14 print("大于 256 的整数情况下的两个变量指针",id(int3),id(int4))
15
16 #大于 0 的浮点数
17 f1=256.4
18 f2=256.4
19 print("大于 0 的浮点数情况下的两个变量指针",id(f1),id(f2))
20
21 #小于 0 的浮点数
22 f1=-2.45
23 f2=-2.45
24 print("小于 0 的浮点数情况下的两个变量指针",id(f1),id(f2))
25
26 #小于-5 的整数
27 n1=-6
28 n2=-6
29 print("小于-5 的整数情况下的两个变量指针",id(n1),id(n2))
```

运行上面的代码后会得到如下的输出：

```
[-5, 256]情况下的两个变量指针 1882718512 1882718512
字符串情况下的两个变量指针 153383856 153383856
大于 256 的整数情况下的两个变量指针 153472560 153472560
大于 0 的浮点数情况下的两个变量指针 153023472 153023472
小于 0 的浮点数情况下的两个变量指针 153023616 153023640
小于-5 的整数情况下的两个变量指针 153471792 153471824
```

读者可以对照着具体的代码情况和打印的指针结果来理解 4.3.8 小节中“2.缓存的重用机制”中介绍的规则，以加深对该部分知识的体会。

缓存规则在不同的代码块中也会有不同的表现。

在下面的代码中，先添加一个函数 `fun`。这个函数 `fun` 的函数体就是一个新的代码块。在函数 `fun` 之外的代码也是一个代码块。在 `fun` 的函数体内，定义一个与 `fun` 之外代码块相同名字的变量（如 `int1`），以测试下同名变量在不同代码下的缓存规则。

接着又调用了变量 `int2`。`int2` 在改代码块中没有定义，系统会去外层代码块中寻找是否有该变量。如果有，则直接访问（访问的就是第一个代码块中的 `int2`）；如果没有，则会报错。具体代码如下：

代码 4-4：缓存机制演示（续）

```
30 def fun():
31     #[-5,256]
32     int1=-5
33     print("fun 中-5 的指针",id(int1),id(int2))
34
35     #字符串类型
36     s1="3344"
37     print("fun 中 3344 字符串的指针",id(s1),id(s2))
38
39     #>256
40     int3=257
41     print("fun 中 257 的指针",id(int3),id(int4))
42
43     #浮点类型
44     f1=256.4
45     print("fun 中 256.4 的指针",id(f1),id(f2))
46
47     #<-5
48     n1=-6
```

```
49     print("fun 中-6 的指针",id(n1),id(n2))
50
51 fun()
```

运行程序，输出如下：

```
fun 中-5 的指针 1882718512 1882718512
fun 中 3344 字符串的指针 153383856 153383856
fun 中 257 的指针 153472368 153472560
fun 中 256.4 的指针 153023736 153023640
fun 中-6 的指针 153471760 153471824
```

根据打印结果可以看出：

- 对于-5 与字符串“3344”的输出，无论是在同一个代码块，还是不同的代码块，它们都是有相同的缓存内容。
- 对于 257 与 256.4 的输出，如在同一个代码块中，则指针是一样的；如在不同的代码块中，则指针不同。
- 对于-6 的输出，每次都都不一样。表明没有对其缓存进行操作。

4.3.10 布尔型关系的运算符

布尔型数值之间需要进行逻辑关系运算。在计算机底层的逻辑电路中，布尔型运算应用得更广泛。Python 同样支持布尔型的运算。在需要使用多个布尔变量联合判断的结果作为条件的场景中，常常会用到布尔型关系运算。

1. 布尔型关系运算符介绍

Python 中所支持的布尔型关系运算符见表 4-5。

表 4-5 布尔型关系运算符

| 运 算 符 | 描 述 |
|---------|----------------------------|
| and | 取“与”。左右都为 True，结果才为 True |
| Or | 取“或”。左右有一个为 True，结果就为 True |
| not 或 ! | 取“反”。如果是 True 结果就为 False |

还需要额外补充几点：

- and 是一个短路运算符。它只有在第一个运算数为 True 时，才会计算第二个运算数的值。

- or 是一个短路运算符。它只有在第一个运算数为 False 时，才会计算第二个运算数的值。
- not 的优先级比其他类型的运算符现低。not a == b 相当于 not (a == b); 而 a == not b 是错误的。

2. 布尔型关系运算符举例

下面通过一个例子来演示布尔型关系运算符的使用方法：

```
a=3                                #定义 3 个变量
b=5
c=None
print(a<5 and c is None)          #使用 and 运算符来进行布尔型的关系运算，输出：True
```

这段代码是以 and 运算符为例来演示的。在执行最后一句代码时，内部顺序如下：

- (1) 运算 a<5，得到结果 True。
- (2) 运算 c is None，得到结果 True。
- (3) 运算 True and True，得到结果 True。

4.3.11 位运算符

位运算符属于更细微的操作，它是针对每个字节中的每个位（bit）进行操作的。位运算符见表 4-6。

表 4-6 位运算符

| 运 算 符 | 描 述 |
|-------|--------|
| & | 按位“与” |
| | 按位“或” |
| ^ | 按位“异或” |
| ~ | 按位“取反” |
| << | 按位左移 |
| >> | 按位右移 |

位运算符在网络通信、硬件驱动领域应用得比较广泛。在编写应用层程序时，用“位”来存储取值范围小的整型变量，可以一定程度地节省内存空间。

4.3.12 实例 7：演示“位运算符”的使用

下面通过一段程序来演示“位运算符”的使用。

实例描述

定义两个变量，分别按照表 4-6 中的运算符对其运算，观察运算结果。

定义两个变量 a 和 b，其中 a 的值为 1，b 的值为 3，按照表 4-6 中的赋值运算符，对其进行“与”“或”“异或”等操作。代码如下：

代码 4-5： 位运算符的使用演示

```
a=1
b=3

print(a&b)           #按位“与”，输出：1
print(a|b)           #按位“或”，输出：3
print(a^b)           #按位“异或”，输出：2
print(~a)            #按位“取反”，输出：-2
print(a<<b)          #按位左移，输出：8
print(a>>b)          #按位右移，输出：0
```

上面的代码每行后面都配有该代码输出的结果。读者可以对比代码和输出结果体会每个操作符的含义。

4.4 strings（字符串）类型

字符串类型属于 Python 中“序列”（sequence）类型的一种。

“序列”类型可以理解成由一个容器和容器中的元素组成的结构体，其中的元素是有一定顺序的。

字符串类型像一个数组，按照一定顺序放置着一个个的字符，如图 4-1 所示。

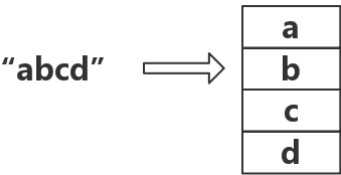


图 4-1 字符串

4.4.1 字符串的描述

字符串大体可以分为两类：

- 单行字符串：使用单引号（' '）、双引号（" "）来表示。
- 多行字符串块：也叫多行字符串，用三个（双或单）引号来表示。如：`"""xxx"""`。

1. 单行字符串

单行字符串表明引号内的字符串必须是单行。如果隔行了，需要用反斜杠“\”符号连接。

例如：

```
a='line1line2'          # a 为单行字符串，内容必须在同一行
a='line1\               # a 为单行字符串的另一种写法。如果隔行了，需要用\来连接下一行
line2'
```

单行字符串的输出也是单行的，如，`print(a)`得到的字符串为：`line1line2`。

2. 多行字符串

在多行字符串的表述中，每行之间可以直接用回车符分开。输出也是按照代码中的回车符号来换行的。例如：

```
#b 为多行字符串
b='''line1
line2
line3'''
```

多行字符串的输出也是多行，如，`print(b)`输出的结果为：

```
line1
line2
line3
```

需要注意的是：定义多行字符串时，千万不要把注释写在字符串定义符的中间，否则会把注释也当成字符串了。例如：

```
b='''line1      #b 为多行字符串
line2
line3'''
```

`print(b)`输出的结果就变成：

```
line1      #b 为多行字符串
```

```
line2
line3
```



提示：

单行字符串在逻辑处理方面用得比较多。多行字符串常用作大段的函数说明，或是对文件或类的说明，有时也会用作以块的方式注释代码。

除了直接定义单行字符串和多行字符串之外，也可以使用 `str` 函数将其他类型转换成字符串。例如：`str(5)`就是可以得到一个字符串“5”。

4.4.2 转义符

在 4.3.1 小节中提到了一个反斜杠“\”符号，它可以将多个单行连成一个单行。这个符号叫作续行符，表示这一行是上一行的延续。还可以使用“`"""....."""`”或者“`"""....."""`”实现多行跨越。

1. 转义符介绍

转义字符是一种特殊的字符。它在代码中看得见，但在输出时会被转换成特殊意义。Python 中有很多转义字符，具体见表 4-7。

表 4-7 转义字符

| 转义字符 | 描 述 |
|---------|----------------------------|
| \（在行尾时） | 续行符 |
| \\ | 反斜杠符号 |
| \' | 单引号 |
| \" | 双引号 |
| \a | 响铃 |
| \b | 退格（Backspace） |
| \e | 转义 |
| \000 | 空 |
| \n | 换行 |
| \v | 纵向制表符 |
| \t | 横向制表符 |
| \r | 回车 |
| \f | 换页 |
| \oyy | 八进制数 yy 代表的字符。例如：\o12 代表换行 |

续表

| 转义字符 | 描 述 |
|--------|----------------------------|
| \xyy | 十进制数 yy 代表的字符。例如：\x0a 代表换行 |
| \other | 其他的字符以普通格式输出 |

2. 原字符串（raw string）

转义字符并不是在所有情况下都有用的。有时还需要输出含表 4-7 中的字符，即，不想让表 4-7 中的字符发生转义。这时需要在字符串前面加一个“r”或者“R”，将其变为原字符串（raw string）。原字符串的内容会与代码中的内容完全一致。例：

```
aa= 'line\tline'      #aa 里面含有转义字符\t，在第二个 line 之前
print(aa)              #将 aa 输出：line→line（→为横向制表符，由 Tab 键输出）
bb= R'line\tline'      #bb 里面含有转义字符\t，同时前面加一个 R，关闭转义功能
print(bb)              #将 bb 输出：line\tline
```

也可以使用函数 repr(str)将字符串 aa(含有转义字符的字符串)转换成原字符串(raw string)。例如：

```
aa= 'line\tline'      #aa 里面含有转义字符\t
print(repr(aa))        #将使用 repr 函数将 aa 的原始字符串输出：
line\tline
```

如果变量 aa 里含有转义字符“\t”，在直接使用 print 函数输出时，会将里面的“\t”转变为制表符；如通过 repr 函数转化后再使用 print 函数输出，则转义字符“\t”停止转义，直接被输出了。

3. 原字符串（raw string）的原理

函数 repr(str)与在字符串前加上“r”或“R”的原理相似，都是在字符串 str 中查找反斜杠“\”字符。如果能找到，就在该字符前面再加一个反斜杠“\”，组成两个反斜杠字符“\\”。根据表 4-7 中的内容，两个反斜杠字符“\\”生成的字符串会转义成一个反斜杠字符“\”，这样就会把原来 str 中那个不需要转义的反斜杠“\”输出来了。

- 在字符串前面加上“r”或“R”，是直接作用在字符串常量上，相对不容易出错。
- 函数 repr(str)是直接作用在字符串变量上。

在使用 repr(str)函数时要注意：输入的字符串参数必须是一个正常的字符串。如果对一个原字符串（raw string）进行 repr 转化，则会使反斜杠变得更多。例如：


```
bb= R'line\tline'          #定义一个原字符串
print(repr(bb))            #对原字符串进行 repr, 输出: 'line\\tline'
```

上例中将原字符串（raw string）传入 `repr` 函数中，生成的结果中出现了两个反斜杠字符，这是我們不想看到的。所以在使用 `repr(str)` 函数时，一定要明确传入参数的字符串类型不能是原字符串。

关于函数 `repr` 的更多内容可以参考后面 9.6.1 小节的“注意”中的内容。

4. 原字符串转成转义字符串

前面介绍了转义字符串变量可以通过 `repr(str)` 函数变为非转义字符串变量。这里再介绍，通过 `eval(str)` 函数将“非转义字符串变量”转变成“转义字符串变量”的方法。见下面的例子：

```
bb= R'line\tline'          #定义一个原字符串
print(eval("'" + bb + "'")) #在原字符串变量前后都加上单引号字符, 通过 eval 函数即可生成转义字符
#输出: line   line
```

5. 规避转义字符带来的问题

转义字符的存在一定程度地简化了编码的复杂度，但在某些情况下也容易带来问题。

例如，当使用一个字符串来表示某个磁盘上的文件路径时（如下行代码），则会产生错误。

```
path = "c:\temp.txt"
```

这时，如果使用函数 `open` 来打开指定的文件（如下行代码），则会报错。系统会将“`\t`”进行转义，认为要打开的文件路径为：“`c: emp.txt`”，于是会提示找不到该文件。

```
open(path)
```

这种情况的解决办法是，将路径写成如下的方式以避免错误：

```
path = r"c:\temp.txt"
```

或

```
path = "c:\\temp.txt"
```

或

```
path = "c:/temp.txt"
```

6. 字符串转化扩展

前面介绍了通过在字符串前面加上一个“r”或“R”，将字符串转成原字符串（raw string）。类似的用法还有很多：

- 在字符串前面加上“b”，将字符串转成二进制字符串。
- 在字符串前面加上“u”，将字符串转成 Unicode 编码的字符串。



提示：

Unicode（统一码、万国码、单一码）是计算机科学领域里的一项业界标准，包括字符集、编码方案等。想要了解更多可以参考百度百科：<https://baike.baidu.com/item/Unicode/750500?fr=aladdin>。

4.4.3 屏幕 I/O 及格式化

print 函数是用来将指定的内容以字符串的形式输出到屏幕上，属于屏幕 I/O 方式的一种。另外，还有与之对应的输入函数 input，它的作用是将键盘的按键信息输入到系统。关于屏幕 I/O 的使用，不仅仅涉及函数本身的参数，还有字符串格式化方面的知识。下面就来学习。

1. 屏幕输出函数 print

print 语句的参数如下：

```
print([object, .....][, sep=' '][, end='newline_character_here'][, file=redirect_to_here])
```

方括号内是可选的，具体参数说明如下。

- Object：要输出的内容。
- sep：分割符。
- end：结束符。
- file：重定向文件（默认值为屏幕输出）。

通过调整 print 函数中字符串的格式，可以控制屏幕上输出字符串的样子。这个调整的过程就叫作字符串的格式化。

字符串格式化常在 print 函数中使用。在编程过程中，还会有很多场景需要用到字符串格式化，例如从屏幕输入、将字符串写入文件等。

- 字符串格式化的方法以下两种。
- 手动拼接：简单地用加号将不同的字符串连接起来。
 - 使用占位符的方法：相对比较高级，应用也比较广泛。主要是先制定一个模板（这个模板就是规定好的格式），在这个模板中某个或者某几个地方留出空位(用占位符来代替)，然后在那些空位（占位符对应的位置）上填入具体内容。

2. 占位符

占位符在一个字符串中占据着一个位置，在输出时将这个位置替换成具体的内容。而占位符并不是随意地替任何内容占位，它有着严格的规则。即，每一个占位符只能替一种特定的类型占位。在字符串中需要选择相应的占位符来替具体的内容占位。

例如代码：`"Hello %s" % "world"`将会得到一个“Hello world”字符串。其中的“%s”就是一个占位符，代表“%s”的位置要用后面的一个字符串来代替。更多的占位符见表 4-8。

表 4-8 占位符

| 占 位 符 | 说 明 |
|-------|--------------------------------------|
| %s | 字符串 |
| %r | 非转义功能的字符串 |
| %c | 单个字符 |
| %b | 二进制整数 |
| %d | 十进制整数，一般会写成%nd，其中 n 代表输出的总长度 |
| %i | 十进制整数 |
| %o | 八进制整数 |
| %x | 十六进制整数 |
| %e | 指数（基底写为 e） |
| %E | 指数（基底写为 E） |
| %f | 浮点数，一般会写成%m.nf，其中 m 代表总长度，n 代表小数点后几位 |
| %F | 浮点数，与上相同 |
| %g | 指数（e）或浮点数（根据显示长度） |
| %G | 指数（E）或浮点数（根据显示长度） |

表 4-8 中的浮点型占位符相对复杂，有必要进行详细说明。

%m.nf 这种形式的占位符，m 代表设定的总位数，n 代表设定的小数点后的位数。当然，具体的输出结果还要根据生成的浮点型数字本身的位数来确定。具体有如下几种情况：

- 输出的浮点型长度小于总长度 m 时，则会在前面补空格。
- 输出的浮点型小数点后的位数小于 n 时，则会在小数点后面补 0。
- 当总长度 m 小于实际整数长度时，则会保存数据完整性，令总长度 m 失效，输出结果右对齐。
- 当总长度 m 小于实际整数加上要输出的小数长度之和时，则会保存数据完整性，令总长度 m 失效，输出结果右对齐。

将这几种情况分别用如下代码演示：

```
print("总长为 8，小数点后为 2，实际长度不足，需要前补空格\n输出：%8.2f"%23.45)
print("总长为 8，小数点后为 4，小数点后位数不足，会在小数点后面补 0\n输出：%8.4f"%23.45)
print("总长为 2，小数点后为 0，总长度比实际整数长度还小，总长度失效\n输出：%2.0f"%223.45)
print("总长为 6，小数点后为 4，总长度 6 小于实际长度 7，总长度失效\n输出：%6.4f"%23.45)
```

上面的代码中，在字符串后面加一个“%”，表示将“%”后的内容填入前面的占位符内。运行后输出如下结果：

```
总长为 8，小数点后为 2，实际长度不足，需要前补空格
输出：□□23.45
总长为 8，小数点后为 4，小数点后位数不足，会在小数点后面补 0
输出：□23.4500
总长为 2，小数点后为 0，总长度比实际整数长度还小，总长度失效
输出：223
总长为 6，小数点后为 4，总长度 6 小于实际长度 7，总长度失效
输出：23.4500
```

上面输出的结果中，□表示一个空格。在计算实际长度时，小数点也占一位。读者可以对比代码中的 m 和 n 的值和具体输出，来理解上面的规则。

3. 手动拼接的格式化

前文说到格式化可以使用手动拼接或占位符的方式来实现。下面来演示具体用法。

先看一个手动拼接的例子：

```
x=5                #定义个整型数 5
print(":",str(x).rjust(2), str(x*x).rjust(3), end=', ') #占两位，以右对齐的
                                     #方式输出 x 本身；占 3 位，以右对齐的方式输出 x*x；结尾用逗号
print(str(x*x*x*10).rjust(4)) #占 4 位，以右对齐的方式输出 x*x*x，结尾用默认的回车
```

在 `print` 函数中，先是使用了字符串转化函数 `str` 将整型变量 `x` 转成字符串，接着使用了字符串对象的 `str.rjust` 方法将字符串格式化输出。

`str.rjust(n)`的作用是将字符串靠右对齐，其中的参数 `n` 代表输出的长度。

- 如果字符串不足这个长度，则默认在左边填充空格。
- 如果字符串的长度大于 `n`，则令 `n` 失效，并不会截断字符串，而是把字符串全部显示。

类似的方法还有字符串左对齐的方法 `str.ljust`，和字符串居中对齐的方法 `str.center`。

这段代码运行后，会输出如下结果：

```
: 5 25, 125
```

“:” 之后是一个空格分割符；接下来的 5 前面有一个空格补位；然后有一个空格分隔符；再下来的 25 前面也会有一个空格补位；然后是一个逗号；最后的 125 前面同样也出现了一个补位的空格。

4. 使用占位符的格式化

下面使用格式化的方式输出与手动拼接例子一样的字符串：

```
x=5                                     #定义一个整型数 5
print(":", '%2d %3d,%4d'%(x, x*x, x*x*x))#在模板中放置 3 个占位符，并指定输出长度
```

使用占位符的方法是：在模板字符串后面加个“%”符号，在“%”符号后面跟上要替换占位符的内容。

上面的代码执行后，得到如下输出：

```
: 5 25, 125
```

可以看到，输出结果与手动拼接例子中的结果一样。

5. 使用 `str.format` 格式化

使用“%”符号来进行字符串的格式化时，要求占位符出场的先后顺序必须与后面的具体内容相匹配。而使用 `str.format` 方法对字符串格式化时，含有占位符的模板与后面内容间的映射关系可以更加灵活。

(1) 基本用法

`str.format` 的基本用法，见如下代码：

```
x=5                                     #定义一个整型数 5
```

```
print(":", '{0:2d} {1:3d}, {2:4d} {0:4d}'.format(x, x*x, x*x*x)) #将 x、 x*x、 x*x*x
三个数值按照字符串模板的格式输出
```

在 `print` 函数里，字符串模板中的占位符都被加上了一个大括号。每个大括号里的第一项用于维护与后面具体内容的对应关系，其数值与 `format` 函数中元素的索引相对应。这样就不需要让模板里的占位符与后面的具体内容顺序一一对应了。

上面的代码执行后，得到如下输出：

```
:  5  25, 125  5
```

可以看到，第一个数与最后一个数都是 5。这表明 `format` 中，索引为 0 的值 `x`，被引用了两次。使用 `str.format` 方法可以使模板与后面的具体内容间的映射关系更加灵活。

(2) 字符串模板的说明

在字符串模板中，冒号后面的格式为 “[补齐字符][对齐方式][宽度]”，其中说明如下。

- 补齐字符：可以是任意字符，但只能是一个字符。如果没有该项，默认是空格。
- 对齐方式：可以是 “<”（左对齐）、“>”（右对齐）、“^”（居中）。
- 宽度：如果对应的输出是整数，需要写成 “nd” 的形式（`n` 代表总长度）；如果对应的输出是浮点型，需要写成 “m.nf” 的形式（`m` 代表总长度，`n` 代表小数点后面的长度）；如果对应的输出是字符串，需要写成 “n” 的形式（`n` 代表总长度）。

下面用代码举例：

```
print('{0:=>10d}'.format(5))      #右对齐, 输出长度为 10 的整数, 用=填充: =====5
print('{0:&<10.3f}'.format(0.5))   #左对齐, 输出长度为 10 的浮点数, 用&填充: 0.500&&&&&
print('{0:-^10}'.format("hello")) #居中对齐, 输出长度为 10 值的字符串, 用-填充: --hello---
```

(3) 简洁用法

如果不追求具体的显示格式，`str.format` 方法还有更简单的使用方法，如下：

```
x=5                                     #定义一个整型数 5
print(":", '{0} {1}, {2} {0}'.format(x, x*x, x*x*x)) #在模板中直接指定后面具体内容的顺序
```

如上面代码中的注释所述，在模板中直接指定后面具体内容的顺序即可，系统会自动根据变量类型匹配对应的占位符。该代码运行后，输出如下：

```
: 5 25, 125 5
```

由于没有指定格式，只是照原样将模板中的序号翻译成对应变量的值输出。

（4）扩展用法 1——结合列表或元组类型

前面的简洁用法中，在 `str.format` 里面需要放置待输出的具体内容，其实这部分的参数还可以使用一个列表或是元组类型的变量来替换（后面在 4.4 节会介绍列表类型，4.5 节会介绍元组类型），具体示例如下：

```
mylist = [5,25,125]                #定义一个列表变量
print(":", '{0} {1}, {2} {0}'.format(*mylist))  #模板中直接指定后面列表变量里具体元素的
顺序即可
mytuple = (5,25,125)               #定义一个元组变量
print(":", '{0} {1}, {2} {0}'.format(*mytuple)) #模板中直接指定后面元组变量里具体元素的
顺序即可
```

这段代码将 `format` 函数里面的参数分别替换成了列表和元组两个变量。该代码运行后，输出了与前面简洁用法例子中一样格式的字符串，如下：

```
: 5 25, 125 5
```

初学者可以先跳过这部分。待看完后面 4.4 节和 4.5 节的内容后，再来复习该部分内容。

（5）扩展用法 2——结合字典类型

`str.format` 还有更高级的用法，需要配合字典类型（字典类型见 4.8 节），具体示例如下：

```
d = {'x':5, 'xx':25, 'xxx':125}    #定义一个字典
print('x is {x}, xx is {xx}, xxx is {xxx}.'.format(**d))  #在模板中，只需填入字典
里对应的名称即可
```

这次，模板中的大括号里不再是整数的序号，而是具体的字符串。这个字符串要与 `format` 中字典变量里的具体条目名称相对应。系统会根据字典里的条目名称找到对应的值，替换到字符串中去。

该代码运行后，输出如下：

```
x is 5, xx is 25, xxx is 125.
```

6. 键盘输入

下面介绍另一个屏幕 I/O 有关的内置函数 `input`。该函数用于从标准输入读取一个行，并返回一个字符串（去掉结尾的换行符）。

用法举例：

```
s = input("请输入字符，并按 Enter 键结束:")
```

```
print("您输入了字符: %s"%s.strip())    #strip 是一个函数,意思是去掉字符串变量 s 中的首尾空格
```

运行该代码,屏幕会输出如下信息:

请输入字符,并按 Enter 键结束:

这时候系统开始等待我们输入字符,比如输入“hello”并按 Enter 键,屏幕显示:

您输入了字符: hello

`input` 函数默认的输入是字符串类型。在实际情况下,可能会需要用户输入不同类型的数据,该函数并不能直接满足要求。这种情况下,只能手动将输入的字符转成需要的类型数据,然后再使用。



注意:

要养成使用 `strip` 配合输入使用的习惯。

`strip` 函数常用于从外界输入内容到系统内的场景下。对于从键盘输入、从文件读取等操作,一般都要对字符串进行一次 `strip` 操作,以便获得非空格开始的有效字符。

另外,还有两个与 `strip` 类似的函数也需要了解:

- ① `lstrip`, 去掉字符串左边的空格;
- ② `rstrip`, 去掉字符串右边的空格。

4.4.4 实例 8: 以字符串为例,演示“序列”类型的运算及操作

前文介绍过,字符串属于“序列”类型(sequence)的一种。“序列”类型,即有序排列,里面的每个元素通过某种符号分开,按照顺序排成一行。本小节就以字符串为例,介绍“序列”类型的相关特征以及用法。

1. “序列”类型的基本操作

“序列”类型中有个重要的概念——索引。索引是指在序列中某个排名的具体数值。这个排名是从 0 开始的。例如:“hello”中 h 的索引是 0, e 的索引是 1, o 的索引是 4。

在了解完索引的概念之后,就可以学习一下关于“序列”类型的几种基本操作了。

- 连接: 使用“+”运算符将两个序列连接起来。在这里就是将两个字符串的连接,例如:“5”+“hello”就会得到一个“5hello”的字符串。
- 重复: 使用“*”运算符,意义是重复该序列内容。如:“a”*3 就会得到一个“aaa”的

字符串。

- 检索：使用中括号加下标的表示方法（[下标]），下标指的是序列中的索引位置。
 - 当下标为正数时是从左往右的索引顺序，从 0 开始计算。如：s="hello", s[0]为“h”。
 - 当下标为负数时是从右向左的顺序，从 1 开始计算，如：s="hello", s[-1]为“o”。
- 反检索：使用 index 函数来完成与检索相反的功能。即，通过字符返回该字符在字符串中的索引，该索引是按照从左向右排列的。如 s.index('e')，会得到一个 1 的数字，表明 e 在 s 中的索引为 1。
- 切片：使用 “[起始:结束:步长]” 形式表示。意思是在原序列数据中，在开始位置切一下，在结束位置切一下，剩的中间片段就是切片。在中间片段的切片中，还要按照步长指定的间隔来取字符串（一般默认为 1）。切片数据是由开始位置的索引与结束位置的前一个索引所组成的（顾头，不顾尾），如：对于一个 s="hello" 的变量，进行 s[1:4] 的切片操作，所得的结果为“ell”。



注意：

在上面的这个切片实例中，要想取出“ello”字符串，一般会将结束部分的索引设为默认值（即 s[1:]），不建议使用 s[1:5]。

因为“hello”字符串中最大索引为 4，在某些循环的处理方式中，这样写还需考虑数组越界的问题，很容易引起错误。

2. 代码举例

对上述的基本操作一一举例，使用代码来演示具体的用法及效果。

实例描述

通过定义一个“hello”字符串，来代表一个“序列”类型。对其进行连接、重复、检索、切片操作的处理，并通过 print 函数将处理的结果输出。

在这里面使用了一个变量 s，并对 s 赋值，令其等于字符串“hello”。在后续的代码中，s 就代表指定的字符串了。当然，将代码中的变量 s 都换成“hello”，是不影响任何结果的。

代码 4-6：“序列”类型的基本操作

```
s='hello'
s2=' daimayisheng'
#连接
print(s+s2)           #输出: hello daimayisheng
#重复
```

```

print(s*3)                #输出: hellohellohello
#检索
print(s[0],s[1],s[2],s[4])#当索引为正数,方向从左到右,索引从0开始。输出: h e l o
print(s[-1],s[-2],s[-4]) #当索引为负数,方向从右到左,索引从1开始。输出: o l e
print(s[-5] )             #因为是从1开始,所有最后一个是5。输出: h
#print(s[5])              #因为是从0开始,所有最后一个是4。索引5已经超过了字符串的范围,
                           #于是报错输出: IndexError: string index out of range 反检索

print(s.index('e'))        #返回e在s中的索引,该索引是从左向右的顺序。输出: 1
print(s.index('l'))        #当有两个l时,返回第一个。输出: 2
print(s.index('w'))        #因为s中没有w字符。所以报错: ValueError: substring not found
                           #切片

print(s[1:3])              #从第一个还是到第三个,步长不指定默认为1。输出: el
print(s[:3])               #开始位置不指定,默认从第一个字符开始截取,取到第3个。输出: hel
print(s[0:])               #结束位置不指定,默认到最后,即从第一个字符开始截取,一直截取到最后
                           #输出: hello

print(s[:])                #开始与结束都不指定,即从第一个字符开始截取,一直截取到最后。输出:hello
print(s[::2])              #步长为2,即每取一个之后,光标往后移动2个。输出: hlo
print(s[::-2])             #步长为-2,即反方向读取,每取一个之后,光标往前移动两元素。输出: olh
print(s[::-1])             #实现字符串的逆序。输出: olleh
print(s[-2::-1])           #实现字符串的逆序,然后再切片。输出: lleh
print(s[:0:-1])            #实现字符串的逆序,然后再切片。输出: olle
                           #字符串不能被修改

s[3]='3'                   #报错,输出: TypeError: 'str' object does not support item assignment

```

该实例代码的注释中包括了运行后的结果。读者可以根据代码的含义,配合注释来理解“序列”类型的基本操作规则。



注意:

上面代码的最后一行,字符串不能被改变。向一个索引位置赋值会导致错误。

4.4.5 关于切片的特殊说明

切片操作的本质上是重新定义一个字符串常量。该常量是依赖于原有字符串,并通过一定规则产生。它也符合前面 4.3.8 小节中“2. 缓存的重用机制”中所讲述的原理。定义切片时,内存的详细变化见如下代码:

```

s='hello'
print(id(s),id(s[::2]),id(s[:]))    #第一个id为s的内存指针,第二个和第三个为切片的指针

```

该代码运行后,输出的结果为:

```

70068184  147319248  70068184

```

可以看出，同样都是切片，但是第二个 id 是重新生成的，而第三个 id 与原字符串 s 是一样的。s[:]的指针与原字符串“hello”的指针一样。这是因为系统默认使用了缓存的重用机制，没有再为其分配新的内存地址。

4.4.6 字符串的相关函数

Python 中内置了很多字符串操作的相关函数，表 4-9 中列举了一些较为常见的函数。

表 4-9 常见字符串操作函数

| 函 数 | 说 明 |
|------------------|---|
| len() | 求序列长度 |
| in : | 判断元素是否存在于序列中。例如: "ab" in "abcd", 返回 True |
| max() : | 返回字符编码的最大值。例如: max("abcd"), 返回 “d” |
| min() : | 返回字符编码的最小值。例如: min("abcd"), 返回 “a” |
| cmp(str1,str2) : | 比较两个序列值是否相同。若相等, 则返回 0 |
| ord(str) | 返回单个字符的字符编码。例如: ord('我'), 则返回 25105 (默认在 UTF-8 下) |
| chr(number) | 根据某个字符编码返回对应的字符。例如: chr(25105), 则返回'我' (默认在 UTF-8 下) |
| str.split () | 这个函数的作用是将字符串根据某个分割符进行分割。例如: a = "I LOVE Python" print(a.split(" ")) #用空格作为分割 输出: ['I', 'LOVE', 'Python'] 输出的返回值是一个列表 (list) 类型, 关于列表的内容, 后续会介绍 |
| chr.join(list) | split 的逆操作, 即将 list 中的每个元素, 用 chr 字符连接起来。例如: b=['I', 'LOVE', 'Python'] print(" ".join(b)) #将 b 按照空格连起来 输出: I LOVE Python |
| str.title() | 将每个单词首字母转为大写。例如: a = "I love Python" print(a.title()) 输出: I Love Python |

表 4-9 中左列倒数第二行为 join 函数，函数 join 的输入参数是一个 list，代表列表类型的意
思（见 4.4 节）。有关函数 join 的使用方法，请见本书 5.7 节。

字符串的方法还有很多。可以通过 dir 命令来查看。在 IPPython console 中输入：

```
dir(str)
```

则会显示如下内容:

```
[ '__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum',
'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

这些内容就是字符串操作相关的全部函数，这里不再一一介绍，要了解某个具体的含义和使用方法，可以使用 `help` 命令查看。

例：想了解 `str` 的 `isalpha` 函数，可以在 IPython console 中输入如下命令。

```
help(str.isalpha)
```

输出结果如下。

```
Help on method_descriptor:

isalpha(...)
    S.isalpha() -> bool

    Return True if all characters in S are alphabetic
    and there is at least one character in S, False otherwise.
```

4.5 list（列表）类型

`list`（列表）是 Python 中使用最频繁的数据类型。它是一个有序集合，与字符串一样，都属于“序列”类型。

`list` 的描述方法是将其内部元素使用中括号括起来，元素之间使用逗号进行分隔。

需要注意的是，列表中每个元素的类型可以互不相同，并且可以嵌套使用。

例如，下面的代码是合法的：

```
tt='hello'           #定义一个变量tt
li = [1,3,4,tt,3.4,"yes",[1,2]] #列表中放置了整型、变量、浮点型、字符串、嵌套列表
```

```
print(li)                #将其打印出来，屏幕输出[1, 3, 4, 'hello', 3.4, 'yes', [1, 2]]
```

上面的代码中定义一个列表变量 `li`，并将其内容输出到屏幕上。这段代码示范了在列表里可以放置多种类型。其中所放置的变量是不受类型约束的，可以是任意类型。



注意：

在定义 `list` 时，允许定义空列表。可以使用 `li=[]` 来实现一个空列表。

4.5.1 list 的运算及操作

`list` 也是“序列”类型，它与字符串一样，都具有 4.3.4 小节实例中所介绍的连接、重复、检索、反检索、切片这几个基本功能。与字符串不同的是，列表中的元素是可以改变的。

4.5.2 list 的内置方法

`list` 的内置方法见表 4-10。

表 4-10 list（列表）的内置方法

| list（列表）的内置函数 | 描 述 |
|--------------------------------|--|
| <code>list.append(x)</code> | 在尾部增加一个元素，等价于 <code>a[len(a):]=[x]</code> |
| <code>list.extend(L)</code> | 将给定的列表 <code>B</code> 中的元素接到当前列表 <code>a</code> 后面，等价于 <code>a[len(a):]=B</code> |
| <code>list.insert(i, x)</code> | 在给定的索引位置 <code>i</code> 前插入项。 <code>a.insert(len(a), x)</code> 等价于 “ <code>a.append(x)</code> ”，表示在尾部插入 <code>x</code> ；如果要在头部插入 <code>x</code> ，可以使用 <code>a.insert(0, x)</code> |
| <code>list.index(x)</code> | 返回列表中第一个值为 <code>x</code> 的项的索引。如果没有匹配的项，则产生一个错误 |
| <code>list.remove(x)</code> | 删除列表中第一个值为 <code>x</code> 的元素。当 <code>list</code> 中没有 <code>x</code> 时会报错 |
| <code>list.pop(i)</code> | 将指定元素弹出的意思。“弹出”是指返回列表中指定索引 <code>i</code> 的元素，并在列表中删除它。 也可以不指定索引。例如： <code>a.pop()</code> ，表示弹出最后一个元素 |
| <code>list.clear()</code> | 删除列表中的所有项，相当于 <code>del a[:]</code> |
| <code>del list[i 或切片]</code> | 删除变量或是删除列表中指定索引 <code>i</code> 的元素，也可以删除列表中指定的切片。当删除列表中指定索引 <code>i</code> 的元素时，等价于 <code>a.remove(a[i])</code> 或 <code>a.__delitem__(i)</code> ，但效率相对较慢。 <code>del</code> 关键字还可以实现清空列表，例如： <code>del list[:]</code> 为清空列表的所有元素，等同于 <code>list.clear()</code> |
| <code>list.count(x)</code> | 返回列表中 <code>x</code> 出现的次数 |
| <code>list.sort()</code> | 列表排序操作 |
| <code>list.reverse()</code> | 逆序操作，等价于 <code>a[::-1]</code> |

上面的代码运行后会有如下输出：

```
[1, 4, 'hello', 3.4, 'yes', [1, 2]] 151650952
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7] 151621832
None 1722913824
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7] 151650952
None 1722913824
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7, [6, 7]] 151650952
```

从结果可以看出以下结论：

- 使用“+”号连接的列表，是将 list3 中的元素放在 list1 的后面得到的 l2。并且 l2 的内存地址值与 list1 并不一样，这表明 l2 是一个重新生成的列表。
- 使用 extend 处理后得到的 l2 是 none。表明 extend 没有返回值，并不能使用链式表达式。即 extend 千万不能放在等式的右侧。这是编程时常犯的错误，一定要引起注意。
- extend 处理之后，list1 的内容与使用“+”号生成的 l2 是一样的。但 list1 的地址在操作前后并没有变化，这表明 extend 的处理仅仅是改变了 list1，而没有重新创建一个 list。从这个角度来看，extend 的效率要高于“+”号。
- 从 append 的结果可以看出，append 的作用是将 list3 整体当成一个元素追加到 list1 后面，这与 extend 和“+”号的功能完全不同。这一点也是需要注意。

2. 删除操作演示

接下来演示关于 del 的基本用法，见下方代码。

代码 4-7：list 操作示例（续）

```
20 #删除操作
21 print(list1)      #输出 list1 的内容及地址[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7, [6, 7]]
22 del list1[2:5]    #删除 list1 的切片
23 print(list1)      #再次输出 list1 的内容及地址 [1, 4, [1, 2], 6, 7, [6, 7]]
24 del list1[2]      #删除 list1 的索引
25 print(list1)      #再次输出 list1 的内容及地址 [1, 4, 6, 7, [6, 7]]
```

运行后得到如下输出：

```
[1, 4, 'hello', 3.4, 'yes', [1, 2], 6, 7, [6, 7]]
[1, 4, [1, 2], 6, 7, [6, 7]]
[1, 4, 6, 7, [6, 7]]
```

这 3 行输出分别是 list1 的原始内容、删除一部分切片内容、删除指定索引内容。可以看到，del 关键字是按照指定的位置删掉了指定的内容。

3. 删除变量演示

使用 `del` 关键字还需要弄清楚，删除的到底是变量还是数据。

下面通过代码来演示并证明一下删除变量的方法。

代码 4-7： list 操作示例（续）

```
26 l2=list1
27 print(id(l2),id(list1))    #打印地址
28 del list1                  #删除 list1 变量
29 print(l2)                  #l2 还能访问，表明地址指向的数据并没有删
30 print(list1)               #再次输出 list1 的内容及地址 NameError: name 'list1' is not defined
```

运行后得到如下输出：

```
149263880 149263880
[1, 4, 6, 7, [6, 7]]
NameError: name 'list1' is not defined
```

第一行输出的内容是 `l2` 与 `list1` 的地址。该输出验证了前面讲述的赋值操作，因为只是传递内存地址，所以 `l2` 与 `list1` 是相等的。

接下来将 `list1` 删掉，并且打印 `l2`，得到了第二行的输出。从输出可以证明 `l2` 所指向的内存数据还是在的，这表明 `del` 删除 `list1` 时仅仅是销毁了变量 `list1`，并没有删除指向的数据。

4. 删除数据

其实除了删除变量这个例子，其他的删除都是删除数据。接下来再举一个极端的例子——将列表数据全部清空，如下代码：

代码 4-7： list 操作示例（续）

```
31 l3=l2
32 del l2[:]                  #删除 l2 全部内容
33 print(l2)                  #输出 l2 的内容及地址 []
34 print(l3)                  #输出 l3 的内容及地址也是 []，表明数据已经被删
```

运行后得到如下输出：

```
[]
[]
```

从输出内容可以看到，先让 `l3` 与 `l2` 指向同样的内存地址。当 `l2` 被清空后，`l3` 的内容也被

清空了。这表明内存中的数据真正改变了。

5. 总结

在实际过程中，这些内存只是被标为无效，并不是真正地被系统回收进行二次使用。如果想让系统回收这些可用的内存地址，需要引入 `gc` 库，并配合如下代码形式：

```
import gc          #引入 gc 库
del list1          #删除 list1
gc.collect()       #回收内存地址
```

在 4.3.8 小节中的“2. 缓存的重用机制”里介绍过：系统为了提升性能，会将一部分变量驻留在内存中。这个机制对于，多线程并发时程序产生大量占用内存的变量无法得到释放，或者某些不再需要使用的全局变量占用着大量的内存，会导致后续运行中出现内存不足的情况。这时候记得使用 `del` 关键字来回收内存，使系统的性能得以提升。它可以为团队省去扩充大量内存的成本。

4.5.4 列表嵌套

Python 中并没有二维数组的概念，但可以通过列表嵌套达到同样的目的。实例如下：

```
mat = [              #定义一个 list 变量 mat，mat 中的每个元素分别是一个 list
    [1, 2, 3],        #mat 中的每个元素中又嵌套着一个 list，嵌套的 list 中包含有 3 个整型
    [4, 5, 6],
    [7, 8, 9]
]
print(mat)           #将 mat 输出
```

上面的代码运行之后，得到如下结果：

```
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

结果显示出这是一个 3×3 的二维数组。这个例子仅仅只是显示了 `list` 能作二维数组而已。在实际编程中，通常会使用 `numpy` 库里面的有关操作来处理二维或多维数组问题。不推荐使用 `list` 来处理二维数组。`numpy` 库是一个支持 Python 使用的第三方库，提供了很多关于数值变换、矩阵处理相关的函数封装，是 Python 语言应用在人工智能领域使用较广的第三方库之一。

4.5.5 实例 10：使用 `list` 类型实现队列和栈

队列和栈是两种数据结构，其内部都是按照固定顺序来存放变量的。二者的区别在于对数

据的存取顺序：

- 队列是，先存入的数据，最先取出。即，先进先出。
- 栈是，最后存入的数据，最先取出。即，后进先出。

list 类型数据本身的存放就是有顺序的，而且内部元素又可以是各不相同的类型，非常适合用于队列和栈的实现。本例将演示使用 list 类型变量来实现队列和栈。

实例描述

定义一个 list 变量，让它充当队列或栈。

(1) 使用 list 的 insert 方法存入 3 个数据，使用 pop 方法分别读取 3 个数据。观察输出数据的顺序，并与队列的顺序进行比较。

(2) 使用 list 的 append 的方法存入 3 个数据，使用 pop 方法分别读取 3 个数据。观察输出数据的顺序，并与栈的顺序进行比较。

1. 使用 list 实现队列

定义一个 list 变量 queue，使用 insert 方法存入数据。

insert 方法的第一个参数设为 0，代表每次都在最前面插入数据。

读取数据时使用的是 pop 方法，即将 queue 的最后一个元素弹出。这样就形成了先进先出的顺序。

见代码 4-8。

代码 4-8：list 实现对列和栈

| | | |
|----|-----------------------------|-------------------------------|
| 01 | queue = [] | #定义一个空 list，当作队列 |
| 02 | queue.insert(0,1) | #向队列里存入一个整型元素 1 |
| 03 | queue.insert(0,2) | #向队列里存入一个整型元素 2 |
| 04 | queue.insert(0,"hello") | #向队列里存入一个字符型元素 hello |
| 05 | print("取第一个元素",queue.pop()) | #从队列里读取一个元素，根据先进先出原则，输出 1 |
| 06 | print("取第二个元素",queue.pop()) | #从队列里读取一个元素，根据先进先出原则，输出 2 |
| 07 | print("取第三个元素",queue.pop()) | #从队列里读取一个元素，根据先进先出原则，输出 hello |

这段代码的运行结果在 print 语句后的注释中有说明。建议读者看懂书中内容后，在本机独立实现并运行一次，以加深印象。

2. 使用 list 实现栈

定义一个 list 变量 stack，使用 append 方法存入数据。

append 代表每次都在最后面添加数据。读取数据时使用的是 pop 方法，即，将 stack 的最后一个元素取出。这样就形成了后进先出的顺序。代码如下：

代码 4-8：list 实现对列和栈（续）

| | | |
|----|-----------------------------|------------------------------|
| 08 | stack = [] | #定义一个空 list，当作栈 |
| 09 | stack.append(1) | #向栈里存入一个整型元素 1 |
| 10 | stack.append(2) | #向栈里存入一个整型元素 2 |
| 11 | stack.append("hello") | #向栈里存入一个字符型元素 hello |
| 12 | print("取第一个元素",stack.pop()) | #从栈里读取一个元素，根据后进先出原则，输出 hello |
| 13 | print("取第二个元素",stack.pop()) | #从栈里读取一个元素，根据后进先出原则，输出 2 |
| 14 | print("取第三个元素",stack.pop()) | #从栈里读取一个元素，根据后进先出原则，输出 1 |

这段代码的运行结果在 print 语句后的注释中有说明。读者在本机同步运行后，可以试试将每次存入和取出操作后的 stack 内容打印出来，观察其变化。

3. 扩展：使用 deque 结构体实现更高效的队列

前面使用 list 实现队列的例子中，插入数据的部分是通过 insert 函数实现的，这种方法效率并不高。因为每次从列表的开头插入一个数据，列表中所有元素都得向后移动一个位置。

还有一个高效的方法，使用标准库的 collections.deque 结构体，它被设计成在两端存入和读取都很快的特殊 list。同时 collections.deque 结构体也可以实现栈的功能，代码演示如下：

代码 4-8：list 实现对列和栈（续）

| | | |
|----|--------------------------------|-----------------------------------|
| 15 | from collections import deque | #导入 deque 结构体 |
| 16 | queueandstack = deque() | #创建空结构体，既可以当队列，又可以当栈 |
| 17 | | |
| 18 | queueandstack.append(1) | #向结构体里存入一个整型元素 1 |
| 19 | queueandstack.append(2) | #向结构体里存入一个整型元素 2 |
| 20 | queueandstack.append("hello") | #向结构体里存入一个字符型元素 hello |
| 21 | print(list(queueandstack)) | #将结构体内容打印出来，输出：[1, 2, 'hello'] |
| 22 | | |
| 23 | print(queueandstack.popleft()) | #实现队列功能，从队列里取一个元素，根据先进先出原则，输出 1 |
| 24 | print(queueandstack.pop()) | #实现栈功能，从栈里取一个元素，根据后进先出原则，输出 hello |
| 25 | print(list(queueandstack)) | #将结构体内容打印出来，输出：[2] |

代码中演示了：

- 使用 `deque` 函数，创建一个 `deque` 结构体。
- 使用 `append` 函数，向结构体（`deque`）里存入数据。
- 使用 `popleft` 函数，以队列的方式读取数据。
- 使用 `pop` 函数，以栈的方式读取数据。
- 使用 `list` 类型转换，将 `deque` 转成 `list`。

4. 总结

本实例的重点是对队列和栈两个结构体进行介绍，并巩固了对 `list` 内置方法的使用。但这只是一个示例，它实现了队列与栈的最基本功能，用于加深对 `list` 类型的理解。

在实际应用场景中，栈和队列的实现要比这个例子复杂，所有的操作都需要用函数封装起来，并加入一定的安全性检查。在多线程环境下，还需要为某些具体的操作加入同步机制（见第10章）。

4.5.6 实例 11：使用函数 `filter` 筛选列表——筛选学生列表中的偏科学生名单

对于序列数据进行过滤，是个很常用的功能。在 Python 中提供了一个内置函数 `filter`，可以很方便地对序列数据进行过滤。因为该函数是根据自定义的过滤函数进行过滤操作，所以支持更加灵活的过滤规则。下面通过实例来演示。

实例描述

定义一个 `list` 变量，里面放置若干学生成绩的信息（包括语文、英语、数学）。通过编写代码筛选出偏科的学生名单。偏科的规则定义如下：

- （1）有两科成绩在 80 分以上，并且有一科在 60 分以下。
 - （2）有一科成绩在 90 分上，并且另外两科成绩都在 60 分以下。
 - （3）有一科成绩在 90 分以上，并且三科的平均分在 70 分以下。
-

为了实现例子中的功能，代码中用到了几个目前没有学到的知识点。读者可以先了解一下，有个印象，看不懂没关系，后面还会详细讲解。

首先是模拟实现一部分的学生成绩数据，接着编写过滤函数，最后使用 `filter` 对数据进行过滤。

1. 模拟学生成绩

直接使用列表来实现学生成绩的模拟。代码如下：

代码 4-9：筛选学生列表中的偏科的名单

```
01 scores = [("Emma", 89, 90, 59),
02  ("Edith", 99, 49, 59),
03  ("Sophia", 99, 60, 20),
04  ("May", 40, 94, 59),
05  ("Ashley", 89, 90, 59),
06  ("Amy", 89, 90, 69),
07  ("Lucy", 79, 90, 59),
08  ("Gloria", 85, 90, 59),
09  ("Abby", 89, 91, 90),]
```

代码中，列表 `scores` 内部的元素为每个学生的学习成绩。每个元素的表现形式都是由括号括起来的。这种被括号括起来的组合类型，叫作元组。这里只是了解一下就好，在 4.5 节还会详细介绍。

2. 编写过滤函数实现偏科规则的判定

编写函数 `handle_filter`，函数的参数为某一个学生的信息，即 `scores` 里面的一个元素。在函数中，先使用 `sorted` 函数对该学生的三科成绩进行排序。接着通过三个 `if` 语句完成偏科规则的判断。只要符合其中一种，就会让程序返回一个 `True`。如果三种规则都不满足，则返回 `False`。

代码 4-9：筛选学生列表中的偏科的名单（续）

```
10 def handle_filter(a):
11     s = sorted(a[1:])          #对三科成绩进行排序
12     if s[-2]>80 and s[0]<60:    #有两科成绩在 80 分以上，并且有一科在 60 分以下的
13         return True
14     if s[-1]>90 and s[1]<60:    #有一科成绩在 90 分以上，并且另外两科成绩都在 60 分以下的
15         return True
16     if s[-2]>80 and sum(s)/len(s)<60: #有一科成绩在 90 分以上，并且三科的平均分在 70 分以下的
17         return True
18     return False
```

这部分的代码涉及到 `sorted` 函数的调用（`sorted` 属于内置函数，会在“附录 A”中的“3. 序列操作内置函数”中介绍）、`if` 语句的知识（见第 5 章的内容）。读者可以先简单知道这么用，等学到后面自然会明白。

3. 使用 filter 函数对列表数据进行过滤

filter 函数的作用是，循环列表中的每一个元素，依次将元素传入过滤函数中执行。如果返回值为 True，则将其保留；如果返回值为 False，则将其忽略。具体代码如下：

代码 4-9：筛选学生列表中的偏科的名单（续）

```
19 aa = list(filter(handle_filter,scores))
20 print(aa)
```

将 filter 函数的返回值传入 list 函数，是要把 filter 的返回值转化为列表。Filter 中传入了两个参数，第一个为过滤处理函数，第二个为要处理的数据。

代码执行后，可以看到有如下结果输出：

```
[('Emma', 89, 90, 59), ('Edith', 99, 49, 59), ('May', 40, 94, 59), ('Ashley', 89, 90, 59), ('Gloria', 85, 90, 59)]
```

结果是经过代码进行过滤后的学生成绩数据，这就是符合规则过滤出来的偏科学生。



注意：

本例子主要是了解列表的使用场景，以及在解决问题中列表所充当的角色。

里面涉及超出列表以外的知识点，在本书后文均介绍，读者不用急于弄懂。

关于 filter 的更多内容可以参考书中“第6章 函数操作”中的知识。

4.6 tuple（元组）类型

tuple（元组）可以理解为 list（列表）的只读版。它与 list（列表）非常类似，内部元素也是按照一定顺序存储的，每个元素的类型也可以各不相同。两者的不同之处在于，元组的元素不能修改。

4.6.1 tuple 的描述

元组的描述方法是：内部元素使用小括号括起来，元素之间使用逗号来分隔。例如：

```
a = (1991, 2014, 'physics', 'math')    #定义一个元组变量 a
print(a, type(a), len(a))             #将 a 的内容、类型、长度打印出来
```

输出结果如下：

```
(1991, 2014, 'physics', 'math') <class 'tuple'> 4
```

元组的“序列”类型相关操作与 `list` 基本一样，下标索引从 0 开始，支持索引、切片等读取方式，但不支持修改。例如：

```
tup = (1, 2, 3, 4, 5, 6)      #定义一个元组变量 tup
print(tup[0], tup[1:5])      #通过切片访问内部数据，输出：1 (2, 3, 4, 5)
tup[0] = 11                  #不支持修改，该句是非法的，会报错误
```

上面代码中的最后一行，企图修改 `tup` 中第 0 个元素的值为 11，但由于 `tup` 是元组，元素不可更改，所以会报错误。



注意：

虽然 `tuple` 的元素不可改变，但它可以包含可变的对象，比如 `list` 列表，并且可变对象内部的内容也是可以修改的。（见 4.5.3 小节的实例演示）

关于 `tuple` 的定义还有一些特殊的规则：

- 在定义包含 0 个元素的 `tuple` 时，直接使用小括号；
- 在定义包含 1 个元素的 `tuple` 时，需要在元素后面加一个逗号。

例如：

```
tup1 = ()      #定义空元组。
tup2 = (20,)   #tup2 中只有一个元素，需要在元素后添加逗号
```

另外，因为字符串常量与元组都有不可修改的特性，所以字符串常量也可以理解成为一种特殊的元组。

4.6.2 运算及操作

元组与列表内部的元素可是为任何类型，这表明元组与列表都可以互相嵌套，也可以嵌套自己。

元组与字符串操作一致，这里不再表述。

4.6.3 实例 12：演示 tuple 的用法

本例将演示 `tuple` 的用法，并与 `list` 的使用做比较。

实例描述

通过 3 段代码来演示 `tuple` 的使用方法以及与 `list` 的区别：

(1) 分别定义一个空的 `tuple` 与一个空的 `list`，观察二者写法上的区别。

(2) 分别定义一个含有唯一元素的 `tuple` 与一个含有唯一元素的 `list`，观察两对象在定义上的区别。

(3) 将 `list` 嵌入到 `tuple` 中，并分别修改 `tuple` 及 `list` 的内容，看看能发生什么。

1. 分别定义一个空的 `tuple` 与一个空 `list`

定义 `tuple` 与 `list` 的空元素类型，`tuple` 直接使用小括号，`list` 直接使用中括号。代码如下：

代码 4-10：元组与 `list` 对比

```
01 #空元素
02 t=()                #定义一个空的 tuple
03 li=[]              #定义一个空的 list
```

因为 `tuple` 不能修改，所以定义一个空的 `tuple` 几乎是没什么用。但定义一个空的 `list` 确实很常用。使用循环语句填充 `list` 结构时，一般会在循环体外面定义一个空的 `list`。这样，当循环次数为 0 时，代码还可以继续执行，不会因为找不到 `list` 变量而报错。

2. 分别定义一个含有唯一元素的 `tuple` 与一个含有唯一元素的 `list`

定义只有一个元素的 `list`，直接使用中括号包含起来即可。

而定义只有一个元素的 `tuple`，除了用小括号包含外，还必须要在元素后面加一个逗号。

下面的代码中分别演示了定义只有一个元素的 `list` 与 `tuple` 时，在元素后面加逗号和不加逗号两种情况。

代码 4-10：元组与 `list` 对比（续）

```
04 t=(3)                #定义一个元素
05 t1=(3,)              #必须要加逗号
06 li=[3]               #对于 list，加不加逗号都可以
07 l1=[3,]              #对于 list，加不加逗号都可以
08 print(t,t1,li,l1)
```

代码运行后结果如下显示：

```
3 (3,) [3] [3]
```

当定义只有唯一元素的元组时，在元素后面一定要加逗号（如代码 4-10 中的第 5 行）。如果不加逗号（如代码 4-10 中的第 4 行），系统会认为只是定义了一个具体的数（见输出结果的

第一个值 3)。

而定义只有唯一元素的列表时，在元素后面加不加逗号，则没有任何影响。例如代码 4-10 中的第 6 和 7 行，分别在元素后面进行了不加逗号与加逗号操作。但是输出的值都是列表（见输出结果的最后两个值，都为[3]）。

3. 将 list 嵌入到 tuple 中

下面代码定义了一个变量 tt，然后将变量与各种类型的常量放到 tuple 类型的 t1 中，来演示 tuple 内部可以包含各种类型。接着修改 t1 中的 list，然后再修改该 t1 中第 0 个元素。代码如下：

代码 4-10：元组与 list 对比（续）

| | | |
|----|---------------------------------------|---------------------------|
| 09 | #元素修改处理 | |
| 10 | tt='hello' | #定义一个变量 tt |
| 11 | t1 = (1,3,4,tt,3.4,"yes",[1,2],(4,3)) | #列表中放置了整型、变量、浮点型、字符串、嵌套列表 |
| 12 | print(t1) | #将其打印出来 |
| 13 | t1[6][0]="3445" | #为元组中的 list 里的元素赋值 |
| 14 | print(t1) | #打印内容 |
| 15 | t1[0]=3 | #改变元组中的元素，返回错误 |

运行后，输出结果：

```
(1, 3, 4, 'hello', 3.4, 'yes', [1, 2], (4, 3))
(1, 3, 4, 'hello', 3.4, 'yes', ['3445', 2], (4, 3))
Traceback (most recent call last):

  File "<iPython-input-9-d7fba5548f53>", line 1, in <module>
    runfile('D:/Python2/4-5 元组与 list.py', wdir='D:/Python2')
.....
TypeError: 'tuple' object does not support item assignment
```

结果中第 1 行显示的是原始 tuple 内容。

第 2 行显示的是 tuple 中的 list 被修改后的内容。

没有任何报错，表明 tuple 中的 list 内容是可以修改的。

在执行代码 15 行时报错了，显示的错误信息是：tuple 对象不支持内部元素的赋值。

4. 总结

该实例证明了 tuple 是 list 的只读版本。从功能的角度来说，list 可以完全地替代 tuple。但

是在某种场合下，`tuple` 还会有它特殊的作用。例如，作为某个函数的返回值时，它可以保证返回的内容不被修改，从而增加了代码的健壮性。

回顾前面 4.3.3 小节中的“5 演示使用 `str.format` 的格式化方法”里面的“（3）扩展用法 1-结合列表或元组类型”的例子，应该明白其中变量“`mylist`”和“`mytuple`”的意义了。



提示：

`format` 函数的参数中，变量“`mylist`”和“`mytuple`”的前面加了个符号“`*`”。这个“`*`”的意思是解包参数列表，即将 `tuple` 或 `list` 中的内容解包出来，作为参数传入到函数 `format` 中。

其实在这种应用场景下，直接使用 `tuple` 的代码就会比使用 `list` 的代码有更强的健壮性。

4.7 set（集合）类型

Python 还有一个数据类型——`set`（集合）。它的主要作用是，进行成员关系测试和消除重复元素。在数据清洗领域运用得比较广泛。

4.7.1 set 的描述

`set` 是一个无序、不重复元素的集合。它的描述方法是：内部元素使用大括号括起来，元素之间使用逗号进行分隔，里面的元素同样可以是任何类型。例如：

```
myset = {'hello', 'hello', 'Python', 'tensorflow', 2, 1, 2} #使用大括号方法定义一个 set（集合）
print(myset)                                              #生成的 set 会自动去掉重复的元素
                                                         #输出: {'hello', 1, 2, 'Python', 'tensorflow'}
```

还可以使用 `set` 函数将其他类型的变量转成 `set`（集合）。例如：

```
mylist = ['hello', 'hello', 'Python', 'tensorflow', 2, 1, 2] #定义一个 list
mytuple = ('hello', 'hello', 'Python', 'tensorflow', 2, 1, 2) #定义一个 tuple
myset = set(mylist)                                           #使用 set 函数将 list 转成 set（集合）
print(myset)                                                  #生成的 set 会自动去掉重复的元素
                                                         #输出: {'hello', 2, 'Python', 'tensorflow', 1}

myset = set(mytuple)                                          #使用 set 函数将 tuple 转成 set（集合）
print(myset)                                                  #生成的 set 会自动去掉重复的元素
                                                         #输出: {'hello', 2, 'Python', 'tensorflow', 1}
```

正常情况下，要定义 `set` 类型变量，使用大括号或 `set` 函数都可以。

但要想定义一个空的 `set` 变量，就必须使用 `set` 函数。

当使用大括号的方法时，如果里面内容为空，则系统会认为这是个“字典”类型（字典类型会在 4.8 节介绍），而不会去创建“集合”类型。

4.7.2 set 的运算

set 的运算与数学中的集合运算相似，支持差、并、交、等操作。

另外，还有一个 in 的语法，用来测试某个元素是否在某个集合里，返回一个布尔型的结果。

下面通过例子来具体演示：

```
helloset = set('hello')          #通过 set 函数，将字符串 “hello” 转成一个集合
tensorflowset = set('tensorflow') #通过 set 函数，将字符串 “tensorflow” 转成一个集合
print('w' in tensorflowset, tensorflowset) #判断 “w” 是否在集合 tensorflowset 中，并打印 tensorflowset 内容

#输出: True {'t', 's', 'l', 'r', 'e', 'f', 'w', 'o', 'n'}
print('w' in helloset, helloset) #判断 “w” 是否在集合 helloset 中，并打印 helloset 内容
#输出: False {'e', 'l', 'o', 'h'}
print(helloset - tensorflowset)  #计算集合 helloset 与 tensorflowset 的差集
#helloset 中有而 tensorflowset 中没有。输出: {'h'}
print(helloset | tensorflowset)  #计算集合 helloset 与 tensorflowset 的并集
#既包括 helloset，又包括 tensorflowset
#输出: {'t', 's', 'l', 'r', 'e', 'f', 'w', 'o', 'n', 'h'}
print(helloset & tensorflowset)  #计算集合 helloset 与 tensorflowset 的交集
#helloset 中有，而 tensorflowset 中也有。输出: {'e', 'l', 'o'}
print(helloset ^ tensorflowset)  #计算集合 helloset 与 tensorflowset 的对称差集
#该集合中的元素在 helloset 或 tensorflowset 中，但不会同时出现在
# helloset 和 tensorflowset 中。输出: {'t', 's', 'r', 'f', 'w', 'n', 'h'}
```

4.7.3 set 的内置方法

set 还有一些自身的内置方法，表 4-11 中列出了 set 中的常见内置函数的意义及用法。

表 4-11 set 的内置方法

| 集合的内置函数 | 描 述 |
|------------------|---|
| set.add(x) | 往集合里添加一个元素 x |
| set.update(list) | 输入参数是一个列表，将列表里的元素全部添加到集合里 |
| set.remove(x) | 删除集合里的元素 x。当 x 不在集合里时，会报错误（输出 KEYERROR） |
| set.discard(x) | 如果结合里有元素 x 时，删除集合里的元素 x |

续表

| 集合的内置函数 | 描 述 |
|--|--|
| set.clear() | 清空集合中的所有元素 |
| set.pop() | 随机选择集合中的一个元素，并删除 |
| len(set) | 返回集合中元素的个数 |
| in | 判断某元素是否在集合里，返回 bool 类型。例如： s=set("ab") print('a' in s) 运行该代码会输出：True |
| not in | 判断集合里没有某元素，返回 bool 类型 |
| set.issubset(set2) 或 set <= set2 | 判断 set 是否是 set2 的子集。返回 bool 类型 |
| set.issuperset(set2) 或 set >= set2 | 判断 set2 是否是 set 的子集。返回 bool 类型 |
| set.union(set2) 或 set set2 | 计算 set 与 set2 的并集 |
| set.intersection(set2) 或 set & set2 | 计算 set 与 set2 的交集 |
| set.difference(set2) 或 set - set2 | 计算 set 与 set2 的差集 |
| set.symmetric_difference(set2)或 set ^ set2 | 计算 set 与 set2 的对称差集 |

4.7.4 不可变集合

Python 中还有一个不可变集合（frozenset），它类似于元组与列表的关系。

在取值上，不可变集合与集合是一样的。不同的是，frozenset 里的值不能被改变。

例如：

```
mytuple = ('hello', 'hello', 'Python',2,1,2)      #定义一个 tuple
myset = frozenset(mytuple)                        #使用 set 函数将元组转成 set( 不可变集合 )
print(myset)                                     # frozenset({1, 2, 'hello', 'Python'})
```

代码中的第 2 行，使用函数 frozenset 将元组 mytuple 转换成了不可变集合 myset。

从最后一行的输出可以看到，myset 里面的元素被去重了，而且会有个明显的标识 frozenset，这表明该集合是不可变的。

4.8 dictionary（字典）类型

字典（dictionary）是 Python 中非常有用的内置数据类型。

字典是一种映射类型（mapping type），它是一个无序的集合。内部元素是键值对形式出现，

即一个关键字（key）与一个值（value）的组合（“键-值”对）。关键字（key）必须是不可变类型，即，list 和包含可变类型的 tuple 不能做关键字。在同一个字典中，关键字必须是唯一的。

4.8.1 字典的描述

字典的描述，与 set 的描述非常地相似，也是用大括号“{ }”括起来的。唯一不同的是，字典的元素必须是“键-值”对（key:value）类型。另外，也可以使用 dict 函数将其他变量（list 或 tuple）转成字典。

例如：

```
mylist = [('hello',1),('good', 2), ['ok', 3]] #定义一个 list 里面嵌套元组和 list，每个
嵌套的元素都是键值对形式
d = dict(mylist)                                #使用 dict 函数将 mylist 转换成字典
d2 = {'hello': 1, 'good': 2, 'ok': 3}           #使用大括号创建字典
print(d,d2)  #输出{'hello': 1, 'ok': 3, 'good': 2} {'hello': 1, 'ok': 3, 'good': 2}
```

定义空字典变量的方法很简单，直接使用大括号就可以，如：mydic = {}。

4.8.2 字典的运算

字典类型的对象要通过关键字（key）来取值，它的用法是在后面加个中括号，里面输入 key 的字符串。

例如：

```
d2 = {'hello': 1, 'good': 2, 'ok': 3} #使用大括号创建字典
print(d2['hello'])                    #取出字典中 key 为 hello 的值，输出：1
```

如果要在字典中更新某个 key 对应的值，直接将其取出来用等号赋值即可。例如：

```
d2 = {'hello': 1, 'good': 2, 'ok': 3} #使用大括号创建字典
d2['hello'] = 'e'                     #将字典中 key 为 hello 的值赋值为“e”
print(d2['hello'])                    #取出字典中 key 为 hello 的值，输出：e
```

上面字典中，key 为“hello”的值，本来为整型 1，却被改成了字符型的“e”。这表明，在键值对中，值的类型是可以被任意修改的。

1. 为字典类型添加元素

如果要在已有的字典中增加一个键值对（key:value），可以直接在后面加个中括号，里面

输入 key 的字符串，并用等号赋值。例如：

```
d2 = {'hello': 1, 'good': 2, 'ok': 3}    #使用大括号创建字典
d2['new'] = 100                          #在字典中加入 key 为 new、值为 100 的键值对
print(d2)                                #输出: {'ok': 3, 'hello': 1, 'new': 100, 'good': 2}
```

从输出可以看到，因为字典中的元素是无序的，所以新加的键值对并不一定是在最后位置。

2. 删除字典类型里的元素

如果要删掉某个键值对，可以使用内置函数 `del`。例如：

```
d2 = {'hello': 1, 'good': 2, 'ok': 3}    #使用大括号创建字典
del d2['hello']                          #在字典中删除 key 为 hello 的键值对
print(d2)                                #输出: {'ok': 3, 'good': 2}
```

3. 获取字典类型里的 key 与 value

字典中还有内置方法 `keys`，可以返回 `dict_keys` 类，里面包含所有的 key；同时还有内置方法 `values`，可以返回 `dict_values` 类，里面包含所有的值。类型 `dict_keys` 和 `dict_values` 都不能直接使用，一般都会将其转成 `list` 来使用。例如：

```
d2 = {'hello': 1, 'good': 2, 'ok': 3}    #使用大括号创建字典
print(type(d2.values()))                 #打印.values()返回值的类型，输出: <class 'dict_values'>
print(d2.values())                       #打印.values()返回值，输出: dict_values([3, 1, 2])
print(type(d2.keys()))                   #打印.keys()返回值的类型，输出: <class 'dict_keys'>
print(d2.keys())                         #打印.keys()返回值，输出: dict_keys(['ok', 'hello', 'good'])
list1 = list(d2.keys())                   #将.keys()返回值转成list类型
print(list1)                             #打印转换后的list，输出: ['ok', 'hello', 'good']
list2 = sorted(d2.keys())                 #由于key的顺序不固定，常用将其转成排序后的list
print(list2)                             #打印排序后的list，输出: ['good', 'hello', 'ok']
```

为了提高代码的健壮性，建议最好要使用 `sorted` 函数来将字典中的 key 转成 `list`。

这里再来看看前面 4.4.3 小节中的“5. 演示使用 `str.format` 的格式化方法”里面的“（4）扩展用法 2-结合字典类型”的例子，其中变量“`d`”就是字典类型。与列表或元组不同的是，将字典类型的参数列表解包传入其他函数调用时，需要用两个星号（**）。所以在 `format` 函数里，变量“`d`”前面加了两个星号（**）。

字典还有更多的内置方法见 4.8.3 小节。

4.8.3 字典的内置方法

字典更多的内置方法见表 4-12。

表 4-12 字典的内置方法

| 字典的内置方法 | 描 述 |
|------------------------------------|---|
| dict.fromkeys(seq [,value]) | 创建一个新字典，序列 seq 中的元素作为字典的键，value（可选）作为字典所有键对应的初始值 |
| dict.get(key[, default=None]) | 返回指定键 key 的值。如果 key 不在字典中，则返回 default 值(默认为 none) |
| dict.setdefault(key, default=None) | 与 get 类似，但如果键不存在于字典中，将会添加键并将键值设为 default |
| del | 删除字典中指定了 key 值的键值对 |
| dict.clear() | 清空字典中的所有元素 |
| dict.items() | 以列表方式返回可遍历的（键，值）元组数组 |
| len(dict) | 返回字典中元素的个数 |
| in | 判断某个 key 是否在字典里，返回 bool 类型 |
| not in | 判断某个 key 不在字典里，返回 bool 类型 |
| dict.keys() | 以列表方式返回一个字典中所有的键 |
| dict.values() | 以列表方式返回字典中的所有值 |
| dict.update(dict1) | 把字典 dict1 的“键-值”对更新到 dict 里。无返回值 |

表 4-12 中几乎列出了字典类型的全部内置函数。有关字典类型的常用函数已在 4.8.2 小节介绍。

剩下的函数与 4.8.2 小节中介绍的函数用法大同小异，读者可以自行尝试。

4.9 对组合对象进行“深拷贝”和“浅拷贝”

组合对象是包含了其他对象的对象，如列表、集合等。在对组合对象赋值时，会涉及一个问题：深拷贝、浅拷贝。下面就来介绍这部分的知识。

4.9.1 浅拷贝

浅拷贝是指创建一个新的对象，其内容是原对象中元素的引用。所有组合对象被创建并赋值的过程，属于浅拷贝。

前面讲过，用等号赋值时，连对象都不会重新创建。只有重新创建对象并为其赋值，才会

发生浅拷贝现象。例如：

```
tt = [[1,2], 'hello']      #定义一个list组合对象
aa = tt                    #直接用等号赋值。只是指针赋值
print(id(tt), id(aa))      #将二者指针打印。输出：203151048 203151048
aa = list(tt)              #使用list创建一个列表对象，并对该对象赋值
print(id(tt), id(aa))      #再次将二者指针打印。输出：203151048 220191304
```

上面的代码最后一行，输出了 `tt` 与 `aa` 的指针。可以看到二者明显不一样，但二者内部的元素指针却没有改变。接着编写代码如下：

```
for x, y in zip(aa, tt):    #将两个list中的每个元素指针打印出来
    print(id(x), id(y))     #输出：1884291568 1884291568
                             1884291600 1884291600
```

可以看到，二者内部元素指针完全没有变化。通过整个例子可以看到 `tt` 浅拷贝得到 `aa`，`aa` 和 `tt` 指向内存中不同的 `list` 对象，但它们的元素却指向相同的对象。（`zip` 函数在 5.3.5 节有详细介绍）

类似的浅拷贝现象还存在于：切片操作、工厂函数、对象的 `copy` 方法、`copy` 模块中的 `copy` 函数。

4.9.2 深拷贝

相对于浅拷贝，深拷贝就很容易理解了。深拷贝是指创建一个新对象并对其赋值时，原对象中的所有元素都会在新对象重新创建一次。

深拷贝是通过 `copy` 模块中的 `deepcopy` 函数来实现的。下面举例：

```
import copy                #导入copy模块
tt = [[1,2], 'hello']     #定义一个list组合对象
aa = copy.deepcopy(tt)     #使用深拷贝将tt赋值给aa
print(id(tt), id(aa))     #打印二者指针。输出：203150536 220189896
for x, y in zip(aa, tt):   #将两个list中的每个元素指针打印出来
    print(id(x), id(y))    #输出：220191752 186994056
                             102740632 102740632
```

可以看到第4行代码中，打印出来的 `aa` 与 `tt` 的指针不一样，这表明系统重新创建了一个 `list`。

在最后两行代码中，遍历 `aa` 与 `tt` 中的元素，依次将其指针打印出来。结果发现列表 `aa` 与 `tt` 的第一个元素指针不相同，但是第二个元素指针却相同。这是因为，二者的第一个元素也是

一个列表（[1,2]），而第二个元素是一个静态字符串“hello”，而 Hello 属于不可变类型的对象。为了提升效率，Python 语言中，在内存里只存一份不可变对象，并将其地址（即引用）赋值给其他变量。

浅拷贝和深拷贝仅仅是对组合对象的操作，而对于“原子”类型（例如数字、字符串等），赋值的操作过程都是直接将引用赋值。

第 5 章

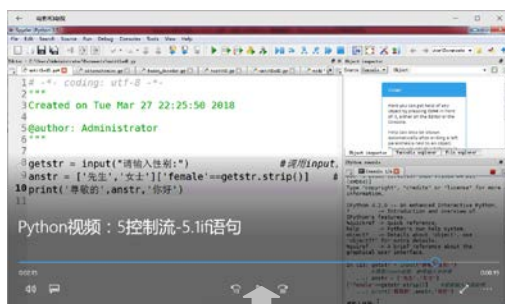
控制流——控制执行顺序的开关

控制流又叫作流程控制，即，根据具体情况来控制程序，令程序执行某些特定的程序块。

Python 中的流程控制语句包括 if 条件语句、while 循环语句和 for 循环语句。还可以细分为 break、continue、pass 等内部流程的控制语句。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解

具体代码操作演示



Python 视频：
5控制流-5.1if
语句.mp4



Python 视频：
5控制流-5.2while
语句.mp4



Python 视频：
5控制流-5.3for
语句.mp4



Python 视频：
5控制流-5.4-5.5
循环控制
语句.mp4



Python 视频：
5控制流-5.6-5.7
列表表
达式.mp4

本章共有 5 段教学视频，总时长为 34 min 左右。包含如下内容：

- 讲述了 if 语句，并通过“实例 13”的演示来强化相关知识。
- 讲述了 while 语句，并演示了“实例 14”中的代码，将十进制转成二进制。
- 首先讲述了 for 语句，接着演示了“实例 15”中使用 for 循环实现冒泡排序，最后介绍了 for 循环与内置函数 range、zip、enumerate 一起使用的知识。
- 讲述了循环过程中的控制语句，包括：break、continue、pass 语句，并演示了“实例 16”的代码，实现一个人机对话中的控制流程。
- 讲述了列表表达式，并通过“实例 17”演示了列表表达式的基本操作。

5.1 if 语句

if 语句是最常用的条件控制语句，关键字有 if、elif、else。

5.1.1 语句形式

一般的表述形式为：

```
if 条件一:
    statements1
elif 条件二:
    statements2
else:
    statements3
```

意思是：

- (1) 如果满足条件一，则执行 statements1 代码。
- (2) 如果不满足条件一，但满足条件二，则执行 statements2 代码。
- (3) 如果条件一和条件二都不满足，则执行 statements3 代码。

这里的条件一、条件二、条件三，分别代表三个条件判断语句。当条件判断语句返回值为 True 时，则代表满足该条件。

5.1.2 演示 if 语句的使用

下面通过代码来演示 if 语句的用法。

```

a = 4          #定义变量 a 的值为 4
b = 5          #定义变量 b 的值为 5
if a>b:        #当 a 大于 b 时;
    c = a      #令 c 等于 a 的值
else:          #否则 c 等于 b 的值
    c = b
print(c)       #将 c 显示。输出: 5

```

在上面代码中，先比较 *a* 与 *b* 的值，当满足 *a* 大于 *b* 的条件，则执行 *c* 等于 *a* 的语句；否则执行 *c* 等于 *b* 的语句。因为 *a* 的值为 4，并不大于 *b* 的值（5），所以将 *b* 的值赋给了 *c*，输出的结果为 5。



注意：

上面的代码中，*if* 与 *else* 部分还有一个简化的写法，即，*c* = [*a*,*b*][*a*<*b*]。这是一个小技巧，可以使代码更为简洁。这种由两个中括号组成的语法，完整的意思是：

- （1）如果第二个方括号里的条件值为假，则返回第一个方括号中的第一个元素。
- （2）如果第二个方括号里的条件值为真，则返回第一个方括号中的第二个元素。

5.1.3 实例 13：根据来访人的性别选择合适的称呼

下面模拟一个人机对话的场景，使用本节前面的知识来控制程序，实现一个动态打招呼的功能。

实例描述

通过一个输入函数来模拟系统接口，获取来访人的性别，并根据输入的性别进行不同的处理：

- （1）如果输入“female”，则显示“尊敬的女士，你好”；
- （2）如果输入“male”，则显示“尊敬的先生，你好”。

使用变量 *getstr* 来接收 *input* 函数得到的键盘输入，根据输入返回来访人性别称呼的字符串，最后将字符串输出。代码如下：

代码 5-1：根据来访性别选择合适的称呼

```

getstr = input("请输入性别:")          #调用 input 函数，获得输入字符串
anstr = ['先生','女士'] ['female'==getstr.strip()] #根据输入返回称呼
print('尊敬的',anstr,'你好')

```

上面代码运行后，输出如下结果：

```
请输入性别：
```

这时，程序会挂起，等待用户输入。通过键盘输入“male”，并按 Enter 键。程序继续执行，输出如下结果：

```
请输入性别: male
尊敬的 先生 你好
```

5.2 while 语句

While 语句用来表述一个循环执行的代码流程。

5.2.1 语句形式

语句的形式为：

```
while 条件一:
    statements1
```

该语句的执行过程，可以分解成如下步骤：

- (1) 执行条件一判断语句，看是否返回 True。
- (2) 如果返回 True，则执行下面的 statements1 代码。
- (3) 执行完 statements1 的代码后，再回到第 (1) 步。
- (4) 如果第 (1) 步返回 False，则整个语句结束。

statements1 属于 while 的子代码块，每一行的开头都需要缩进。

5.2.2 演示 while 语句的使用

下面通过代码演示 if 语句的用法。

```
c=4          #定义一个变量 c
while c>0:   #使用 while 循环。当 c 大于 0，就执行下面语句
    print(c) #输出 c 的值
    c -=1    #c 自身减 1
```

在上面代码中，while 的循环条件是 c 大于 0。在 while 循环体里，每次都会打印出 c 的值，并将 c 自身减 1。代码运行后输出如下：

4
3
2
1

变量 `c` 的初始值为 4。`c` 经过 4 次减 1 操作将会变为 0，不符合 `while` 的条件，于是结束循环。

5.2.3 实例 14：将十进制数转化为二进制数

下面通过实例演示 `while` 的使用。

实例描述

通过输入函数获取一个十进制数值，并将其转换为二进制数并输出到屏幕。

转换的原理是：将输入的十进制数值循环地除以 2，直到返回的结果商为 0 为止。这期间，每次除 2 的余数就是转换后的二进制结果。使用变量 `a` 来接收 `input` 函数接到的键盘输入，并将其转化为整型，然后通过 `while` 循环来处理。代码如下：

代码 5-2：将十进制数转化成二进制数

| | |
|--------------------------------------|--------------------------------------|
| <code>a = input("请输入一个十进制数：")</code> | <code>#获取一个十进制数</code> |
| <code>d=int(a)</code> | <code>#将输入的字符串转为整型</code> |
| <code>s=""</code> | |
| <code>while d!=0:</code> | <code>#使用 while 循环，直到 d 的值为 0</code> |
| <code>d,f=divmod(d,2)</code> | <code>#除以 2，并返回除数和余数</code> |
| <code>s=str(f)+s</code> | <code>#余数作为转换后的二进制数，除数作为循环条件</code> |
| <code>print(s)</code> | <code>#将结果输出</code> |

上面代码运行后，输出如下结果：

请输入一个十进制数：

这时，程序会挂起，等待用户输入。通过键盘输入“6”并按 `Enter` 键。程序继续执行，输出如下结果：

110

结果显示为 110。正是十进制数 6 转化成二进制数的结果。

5.3 for 语句

Python 中的 for 语句与 while 语句都是用来循环执行代码块的。for 循环的执行条件，不是判断逻辑是否为真，而是遍历一个序列容器。

5.3.1 语句形式

for 的语句形式为：

```
for item in 序列数据:
    statements1
```

意思是，每次从序列容器中取一个值（item），然后执行语句 statements1。当遍历完整个序列容器，循环也就结束了。

关于 for 循环和 if 语句的使用，可以参见 5.3.4 小节的实例演示。

5.3.2 在 for 循环中，使用切片

序列容器可以是任何 Python 支持的序列数据类型。

如果在循环体内，执行语句 statements1 对 for 后面的序列数据进行了修改，这会影响到 for 语句的初始循环次数。切片的介绍见 4.3.4 小节。

避免上面问题的方法是：使用切片的方法为该序列数据做一个副本，让 for 来遍历副本中的序列数据，这样即使 statements1 语句修改了原始的序列容器中的数据，也不会影响到初始的循环次数了。例如：

```
words = ['I', 'love', 'Python'] #定义一个列表
for item in words[:]:          #for 后面没有使用 words，而是使用了切片作为 words 的副本
    words.insert(0, item)      #向 words 里插入一个元素
    print(item)                #打印 item，三次迭代分别输出：'I'、'love'、'Python'
print(words)                   #打印整个 words，输出：['Python', 'love', 'I', 'I', 'love', 'Python']
```

上面代码中的第 2 行，使用了 words 的切片作为副本，完成了 3 次迭代（因为有 3 个元素，各迭代一次）。

再来看一个错误的写法。假如不使用副本，将第二句代码换成：

```
for item in words:
```

这将会带来死循环。因为，每次从 `words` 中遍历一个数据，执行下面语句时，就会为 `words` 添加一个数据，这样永远也无法将 `words` 中的数据遍历完。注意，一定要避免这种错误的写法。

5.3.3 在 for 循环中，使用内置函数 range

在 `for` 循环中，还可以使用内置函数 `range` 来遍历一个数字序列。

1. range 介绍

`range` 的意思是返回一个数字区间的所有整数。

(1) 输出大于零的序列

单独的 `print(range(5))` 打印不出 0~5 之间的数字。输出的是 “`range(0, 5)`”，代表从 0~5 的一个范围。

如果要想将其内容散列出来，可以将其转化成 `list` 类型再打印，例如：

```
print(list(range(5))) #将 0~5 间的整数转化成 list 类型，并打印出来。输出: [0, 1, 2, 3, 4]
```

`range` 中的参数默认是从 0 开始。上例中，`range` 的参数 5 代表从 0~5 之间的数。这里 0~5 区间的取值仍然与切片的取值一致，即“要头，不要尾”。意思就是，0~5 区间的数包含起始值（0），但是不包含结束值（5）。

(2) 输出大于零的序列

如果要将比 0 小的数传入 `range`，会打印不出内容。例如：

```
print(list(range(-5))) #-5 比 0 小，range 找不到任何比 0 大的数，只能返回空。输出: []
```

这种情况下，可以通过指定起始值来将 -5~0 之间的数打印出来。例如：

```
print(list(range(-5,0))) #将-5~0 间的整数转化成列表类型，并打印出来。输出: [-5, -4, -3, -2, -1]
```

上例中，`range` 里面的第一个参数 -5 代表起始值，第二个参数 0 代表结束值。

(3) 步长

`range` 函数中还可以有第三个参数——步长。即，从起始到结束，每隔“步长个数字”返回一次。例如：

```
print(list(range(-5,0,2))) #在-5~0 间，每隔两个数取出一个，并转化成 list 类型，打印出来。输出: [-5, -3, -1]
```


2. range 与 for 结合

了解完 range 后，再来看一个 range 与 for 组合的例子：

```
for i in range(5):           #循环遍历 0~5 之间的整数
    print(i)                 #将取出的值打印出来
```

这样就实现了循环 5 次的控制。在 range 中放置一个整数（5）的方式，指定了代码的具体循环次数。上例执行后，输出：

```
0
1
2
3
4
```

5.3.4 实例 15：利用循环实现冒泡排序

在算法编程领域，循环与判断语句的应用非常频繁。下面就来实现一个算法的例子。

实例描述

使用 for 循环对一个列表进行排序。要求不使用内置的库函数。

冒泡排序是数据结构中的经典算法。手动实现冒泡排序，对初学者锻炼自己的编程逻辑有很大帮助。

冒泡排序算法的运作过程如下：

（1）比较相邻的元素。如果第一个比第二个大，就交换它们两个。

（2）从最开始的第一对到结尾的最后一对，对每一对相邻元素做步骤（1）所描述的比较工作，并将最大的元素放在后面。这样，当从最开始的第一对到结尾的最后一对都执行完后，整个序列中的最后一个元素便是最大的数。

（3）将循环缩短，除去最后一个数（因为最后一个已经是最大的了），再重复步骤（2）的操作，得到倒数第二大的数。

（4）持续做步骤（3）的操作，每次将循环缩短一位，并得到本次循环中的最大数。直到循环个数缩短为 1，即没有任何一对数字需要比较。最终便得到了一个从小到大排序的序列。

具体实现的代码如下：

代码 5-3：冒泡排序

```
01 n = [5,8,20,1] #定义一个列表
02 print("原数据: ",n)
03
04 for i in range(len(n)-1): #这个循环负责设置冒泡排序进行的次数
05     for j in range(len(n)-i-1): #j 为列表索引（索引可参见 4.3.4 小节）
06         if n[j] > n[j+1]: #比较两个数
07             n[j], n[j+1] = n[j+1], n[j] #交换
08
09 print("排序后: ",n) #输出结果
```

上面代码使用了两层循环：

- 外层循环，赋值冒泡排序进行的次数，见代码第 4 行。
- 内层循环，负责将列表中相邻的两个元素进行比较，并调整顺序。将较小的数放在前面，较大的数放在后面，见代码第 6、7 行。

程序运行后，输出如下结果：

```
原数据: [5, 8, 20, 1]
排序后: [1, 5, 8, 20]
```

5.3.5 在 for 循环中，使用内置函数 zip

for 语句还可以配合内置函数 zip，以同时遍历多个序列。

1. zip 介绍

zip 函数可以将任意多的“序列”类型数据结合起来，生成新序列数据。生成的新序列数据中，每个元素都是一个元组。该元组的元素是由传入 zip 函数中的多个序列数据的元素组成。例如：

```
x=[1,2,3] #定义两个列表，x 与 y
y=[3,2,"hello"]
t = zip(x,y) #通过 zip 生成一个新的序列 t，这时 t 是 zip 类型
print(tuple(t)) #将 t 转成元组，并打印。输出：((1, 3), (2, 2), (3, 'hello'))
```

当然，生成的序列 t 也可以转成列表（list）类型。例如：

```
x=[1,2,3] #定义两个列表，x 与 y
y=[3,2,"hello"]
t = zip(x,y) #通过 zip 生成一个新的序列 t，这时 t 是 zip 类型
print(list(t)) #将 t 转成列表，并打印。输出：[(1, 3), (2, 2), (3, 'hello')]
```

需要注意的是，当 `zip` 对象 (`t`) 被转化为元组或列表后，就会自动销毁。如果再使用 `t`，将得不到具体的元素。例如：

```
x=[1,2,3]           #定义两个列表，x与y
y=[3,2,"hello"]
t = zip(x,y)         #通过zip生成一个新的序列t，这时t是zip类型
print(list(t))       #将t转成列表，并打印。输出：[(1, 3), (2, 2), (3, 'hello')]
print(tuple(t))      #再次使用t，将t转成元组，得到的是空元组。输出：()
```

`zip` 对象还可以通过前面加个字符“*”的方式来完成 `unzip` 的过程，所谓 `unzip`，就是将 `zip` 生成的数据返回。例如：

```
x=[1,2,3]           #定义两个列表，x与y
y=[3,2,"hello"]
t = zip(x,y)         #通过zip生成一个新的序列t，这时t是zip类型
print(*t)            #在t前面加一个*，完成unzip。输出：(1, 3) (2, 2) (3, 'hello')
print(*t)            #同样，t只能unzip一次，再次使用时，会返回空值。没任何输出
```

上例中，在最后两行特意调用了两次 `zip` 对象 `t`。可以看到第二次调用 `t` 时，得到的输出是空的。这也是值得注意的地方。

如果 `zip` 里面的序列长度不同，就会以最短的序列数据为主。例如：

```
x=[1,2,3,4,5,6]     #定义两个list——x与y，x的长度会更大一些
y=[3,2,"hello"]
t = zip(x,y)         #通过zip生成一个新的序列t，这时t是zip类型
print(list(t))       #将t转成列表。输出：[(1, 3), (2, 2), (3, 'hello')]
```

上例中，`y` 的长度最短，只包含 3 个元素。所以生成的 `t` 也只有 3 个元素。

传入 `zip` 中的类型可以不同。下面演示 `zip` 参数一个是列表和一个是元组的情况：

```
x=[1,2,3,4,5,6]     #定义1个列表x
y=(3,2,"hello")      #定义1个元组y
t = zip(x,y)         #通过zip生成一个新的序列t，这时t是zip类型
print(list(t))       #将t转成列表。输出：[(1, 3), (2, 2), (3, 'hello')]
```

上例中，将列表 `x` 和元组 `y` 一起传入 `zip` 里一样可以得出 `zip` 对象。这表明，`zip` 的参数可以是任意序列类型。

2. zip 与 for 结合

了解完 `zip` 后，再来看一个 `zip` 与 `for` 结合的例子：

```
x=[1,2,3,4,5,6]      #定义1个列表x
y=(3,2,"hello")      #定义1个元组y
for t1,t2 in zip(x,y): #循环遍历 zip 后的 x 和 y
    print(t1,t2)      #将 t1、t2 打印出来
```

for 循环中，直接可以从 zip 对象取出每个迭代的元素，不需要再转成列表或元组。上例执行后，输出：

```
1 3
2 2
3 hello
```

5.3.6 在 for 循环中，使用内置函数 enumerate

在 for 循环中，还可以使用内置函数 enumerate 来遍历一个序列容器。

enumerate 函数的作用是，将“序列”类型的数据生成带序号的新序列数据。

1. enumerate 介绍

在使用 enumerate 生成的新序列中，每个元素都是一个元组，该元组是由传入 enumerate 函数中序列的元素与其对应的索引组成的。例如：

```
x=["hello",5,6]      #定义1个列表x
t = enumerate (x)    #通过 enumerate 生成一个新的序列 t，这时 t 是 enumerate 类型
print(tuple(t))      #将 t 转成元组，并打印，输出：((0, 'hello'), (1, 5), (2, 6))
```

2. enumerate 与 for 结合

了解完 zip 后，再来看一个 enumerate 与 for 组合的例子：

```
x=["hello",5,6]      #定义1个列表x
for i,t2 in enumerate (x): #循环遍历 enumerate 后的 x
    print(i,t2)          #将 i、t2 打印出来
```

for 循环中，需要定义两个变量来接收 enumerate 后的返回值：一个是元素的索引，一个是具体的元素。上例执行后输出：

```
0 hello
1 5
2 6
```

enumerate 与 for 的结合为程序提供了更大的方便性。enumerate 的第一个返回值在循环里同时也起到计数的作用，可以直接当作循环的次数来使用。

5.4 对循环进行控制——break、continue、pass 语句

在循环内部可以使用 `break`、`continue` 和 `pass` 语句，来根据执行语句的具体情况，对循环的过程进行控制。具体意义如下。

- **break**：跳出当前的 `for` 或 `while` 循环。
- **continue**：终止当前这一次执行，进行循环的下一一次迭代。
- **pass**：该语句什么都不做，是为了保持程序结构的完整性。常用在语法上需要一条语句但又不需要任何操作的情况下。

接下来将通过实例来演示这几个语句的使用（见 5.5 节）。

5.5 实例 16：演示人机对话中的控制流程（综合应用前面语句）

下面模拟一个人机对话的场景，通过前面的控制流知识来对程序进行控制。

实例描述

通过一个循环来获得用户的输入，并根据不同的输入进行不同的处理：

- （1）如果输入“hello”，进入主程序，开启人机对话服务。
 - （2）如果输入“go away”或是“Bye”，退出程序。
 - （3）如果输入“pardon”，重新等待用户输入。
-

1. 编写代码

建立一个 `while` 循环，在循环体内获取用户输入，并根据输入的内容进行不同的操作（具体见代码中的注释），代码如下。关于 `break`、`continue`、`pass` 语句，请重点看下面注释中加粗的行。

代码 5-4：人机对话控制流程

```
getstr = ''                                #定义一个空字符串，用来接收输入
while("Bye"!=getstr):                     #使用 while 循环
    if ''==getstr:                          #如果输入字符为空，输出欢迎语句
        print("hello! Password!")
    getstr = input("请输入字符，并按 Enter 键结束:")#调用 input 函数，获得输入字符串
    if 'hello'==getstr.strip():              #如果输入字符串为 hello，启动对话服务
        print('How are you today?')
```

| | |
|------------------------------------|-------------------------------|
| getstr = "start" | #将 getstr 设为 start, 标志是启动对话服务 |
| elif 'go away'==getstr.strip(): | #如果输入的是 go away, 则退出 |
| print('sorry! bye-bye') | |
| break | #使用 break 语句退出循环 |
| elif 'pardon'==getstr.strip(): | #如果输入的是 pardon, 请重新再输出一次 |
| getstr = '' | |
| continue | #continue 将结束本次执行, 开始循环的下一次执行 |
| else: | |
| pass | #什么也不做, 保持程序完整性 |
| if 'start'== getstr: | #如果 getstr 为 start, 启动对话服务 |
| print('...init dialog-serving...') | #伪代码, 打印一些语句, 代表启动了对话服务 |
| print('... one thing...') | |
| print('... two thing...') | |
| print('.....') | |

这里只是模拟人机对话的控制流程, 并没有实现人机对话的真实操作。在人机对话部分, 使用了函数 `input` 来实现输入功能; 在回答部分, 使用了 `print` 函数输出字符串的方式来实现机器的输出功能。

2. 运行程序

代码运行后会显示如下输出:

```
hello! Password!
```

请输入字符, 并按 Enter 键键结束:

程序停在这里, 等待输入。这时输入 “pardon”, 会有如下输出:

```
请输入字符, 并按 Enter 键结束: pardon
```

```
hello! Password!
```

请输入字符, 并按 Enter 键结束:

可以看到代码中的 `if` 语句、`elif` 语句、`continue` 和 `while` 循环发挥了作用。

(1) 先通过 `if` 语句判断输入是否是 “hello”。

(2) 若不是, 则运行 `elif` 语句接着判断。

(3) 一直执行到输入字符串与 “pardon” 相等的 `elif` 语句, 才执行该条件下的 `continue` 语句。

(4) 通过 `continue` 语句结束本次循环, 开始重新迭代。

再次输入“hello”，会有如下输出：

```
请输入字符，并按 Enter 键结束：hello
How are you today?
...init dialog-serving...
... one thing...
... two thing...
.....
请输入字符，并按 Enter 键结束：
```

程序接收到“hello”指令，在内部将 `getstr` 设成了“start”。后面的代码会判断 `getstr` 的值。当 `getstr` 为“start”时，便开始启动人机对话服务。

接着输入“go away”或“Bye”，程序退出，显示如下：

```
请输入字符，并按 Enter 键结束：go away
sorry! bye-bye
```

5.6 利用 for 循环实现列表推导式

列表推导式是一种创建列表的方法。它的应用场景是：当需要对一个序列数据中的每个元素做一些操作，并将操作后的结果作为新列表的元素。

列表推导式提供了从序列创建列表的简单途径，它可以根据指定的条件来创建子序列。

写列表推导式时常会与 `for` 结合在一起。一般会写成：在一个方括号里面写一个表达式，后面再跟一个或多个 `for` 或 `if` 子句。这样会生成一个列表，该列表中的元素就是，一个 `for` 或 `if` 子句中遍历的具体元素经过表达式生成的结果。例如：

```
Y=[1,0,1,0,1,1,1,1,0]          #定义一个list，包含0、1两个元素
colors = ['r' if item == 0 else 'b' for item in Y[:]]    #使用列表推导式将Y中的0变成
“r”，1变成“b”
print(colors)                  #打印 colors 内容，输出：['b', 'r', 'b', 'r', 'b', 'b', 'b', 'b', 'r']
```

上面的代码常常用于设置所绘图中某个点的颜色。假设列表 `Y` 是通过某种运算生成的结果，里面包含了两种结果（0 和 1）。现在要把 `Y` 中的 0 数据用红色（r）表示，1 的数据用蓝色（b）表示。这时就可以使用列表推导式生成一个与 `Y` 对应的颜色列表。

列表推导式还可以生成嵌套列表或元组，例如：

```
m = [[1,2,3], [4,5,6], [7,8,9]]    #定义一个列表
```

```
t = [ (r[0],r[1],r[2]) for r in m ]      #外面的 for 是变量 m 的每一行,里面是将每行的三个元素变成元组
print(t)                                #m 由嵌套列表变成了嵌套元组,输出: [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

上面的第二行也可以写成 `t = [tuple(r) for r in m]`, 是一样的效果。

5.7 实例 17: 利用循环来打印“九九乘法表”

这个例子使用了 5.6 节的知识。

实例描述

使用列表推导式实现一个“九九乘法表”, 并输出。

本例先使用列表推导式生成一个存有“九九乘法表”算式的列表, 并用 `for` 循环将其输出。接着对代码进行优化, 使用字符串的 `join` 方法来代替 `for` 循环输出列表的语句, 使代码语句变得更简洁。

1. 编写代码

建立两个 `for` 循环: 外层循环负责选取计算乘法表中的 1~9 数字, 并生成该数字所对应的乘法表列表; 内层循环负责将列表中的乘法算式打印输出。代码如下:

代码 5-5: 打印九九乘法表

```
01 for x in range(1,10):                #循环 1~9 的乘法表数字
02     l = ['%s*%s=%-2s' % (y,x,x*y) for y in range(1,x+1)]#将某个数字的所有乘法
    式子, 放到列表
03     for i in range(len(l)):           #使用 for 循环打印列表内容
04         print(l[i],end=' ')
05     print('')                         #换行
```

代码运行后, 输出如下结果:

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6  3*3=9
1*4=4  2*4=8  3*4=12  4*4=16
1*5=5  2*5=10  3*5=15  4*5=20  5*5=25
1*6=6  2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7  2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8  2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9  2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```


2. 使用 join 函数代码

这里将使用一个 join 函数来输出列表中的元素。有关函数 join 的说明，请参见 4.3.5 小节中的表 4-9。

代码如下：

代码 5-5：打印九九乘法表（续）

```
06 for x in range(1,10):
07     l = ['%s*%s=%-2s' % (y,x,x*y) for y in range(1,x+1)]
08     print(' '.join(l[i] for i in range(len(l))))
```

第 6、7 行和第 1、2 行一样，第 8 行使用 join 函数将列表 l 中的元素用空格连接起来，完成了输出。代码运行后同样可以显示出“九九乘法表”。这里将不再显示结果。

另外还可以更进一步优化，将最外层的 for 循环也合并进来，变成一行代码。例如：

代码 5-5：打印九九乘法表（续）

```
09 print ('\n'.join([' '.join(['%s*%s=%-2s' % (y,x,x*y) for y in range(1,x+1)])
    for x in range(1,10)]))
```

上面的代码使用了两个 join 函数：第一个 join 函数，将每行的输出用换行符“\n”连接起来；第二个 join 函数，将列表 l 中的元素用空格连接起来。最终实现了用一行代码输出“九九乘法表”的功能。



注意：

在编写代码时，不推荐将所有代码压缩为一行的方式。它会使程序很难调试，也容易隐含一些 bug。

5.8 理解 for 循环的原理——迭代器

for 语句的循环次数，是根据遍历序列容器中的数据个数来决定的。

在遍历的过程中，可以隐式地调用了内置函数 iter。函数 iter 被调用后，会返回一个迭代器对象。迭代器对象定义了__next__方法，在每次访问时会得到一个元素。当没有任何元素时，__next__将通过 StopIteration 异常来告诉 for 语句停止迭代。

具体代码演示如下：

```
x=[1,2,3]          #定义一个列表
it=iter(x)          #调用 iter 函数返回一个迭代器对象
print(it.__next__()) #调用迭代器的__next__方法，返回一个元素。输出：1
print(it.__next__()) #调用迭代器的__next__方法，返回一个元素。输出：2
print(it.__next__()) #调用迭代器的__next__方法，返回一个元素。输出：3
print(it.__next__()) #再次调用迭代器的__next__方法，此时已经没有元素，返回 StopIteration
```

上例中最后一句代码执行时，指针已经走到了列表的最后一个位置。再次获取元素时，发生了异常。在 for 语句中内置的代码会捕获到该异常，然后退出循环。

另外，Python 中还有一个内置函数——next，它是迭代器对象中的__next__ 方法的另一种写法。所以，上面的代码还可以写成如下形式：

```
x=[1,2,3]          #定义一个列表
it=iter(x)          #调用 iter 函数返回一个迭代器对象
print(next(it))      #调用迭代器的__next__方法，返回一个元素。输出：1
print(next(it))      #调用迭代器的__next__方法，返回一个元素。输出：2
print(next(it))      #调用迭代器的__next__方法，返回一个元素。输出：3
print(next(it))      #再次调用迭代器的__next__方法，此时已经没有元素，返回 StopIteration
```

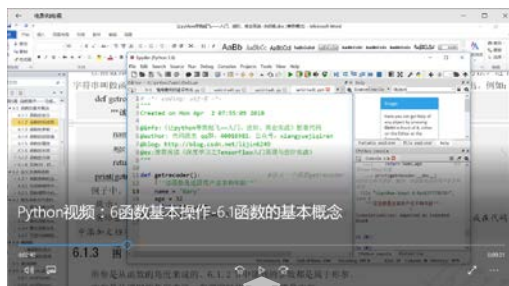
用迭代器访问对象在 Python 中非常常用。除了 for 循环外，还有许多内置函数（例如 sum、min、max 等函数）内部的实现都是使用迭代器访问对象。

函数——功能化程序片段的封装

在面向过程开发中,最重要的是模块化思想。函数在模块化思想中起到了至关重要的作用。它可以将程序拆分成一个个程序片段,以便独立地实现某个完整的功能,进而实现模块化。

函数（function）是组织好的、可重复使用的、具有一定功能的代码段。它能提高模块和代码的重复利用率。在 Python 中，用户可以自定义函数。同时 Python 还提供了很多内置函数（比如 print、input 等）以方便用户调用。

作者按照图书的内容和结构,录制了同步对应的教学视频。



具体代码操作演示



Python视频
函数基本操
-6.10工厂函
mp4

本章共有 8 段教学视频，总时长为 117 min 左右。包含下内容：

- 讲述了函数的基本概念。
- 讲述了定义及调用函数。
- 讲述了匿名函数与可迭代函数。
- 讲述了偏函数和递归函数。
- 讲述了 eval 与 exec 函数。
- 讲述了实例 19，利用 exec 函数进行批量函数调用的例子。
- 讲述了生成器类型和作用域。
- 讲述了工厂函数。

6.1 函数的基本概念

Python 中，可通过固定的格式来定义函数。函数有具体的组成部分（函数名、函数体、参数、返回值）。从分类上来看，函数可分为内置函数、自定义函数等。为了实现不同的编程需求，还可以为函数加上各种规则及作用域的限制，以完成整个功能。

6.1.1 函数的定义

定义函数使用关键字 `def`，后接函数名，再后接放在圆括号 `()` 中的可选参数列表，最后是冒号。格式如下：

```
def 函数名(参数 1, 参数 2, ....., 参数 N):
```

例如：

```
def hello(strname):           #定义一个函数 hello，strname 是代表传入的参数
    print (strname)           #函数的实现，打印参数
```

调用时，直接使用函数名称即可。例：

```
hello("I love Python!")      # 调用函数，这时屏幕输出：I love Python!
```

6.1.2 函数的组成部分

Python 中的有多种不同类型的函数。无论它们的功能差别有多大，其组成部分是相同的。

1. 函数的组成

函数有四个组成部分。

- 函数名：def 后面的名字，例如 6.1.1 小节例中的 `hello`。
- 函数参数：函数名后面的变量，例如 6.1.1 小节例中的 `strname`。
- 函数体：函数名的下一行代码，起始需要缩进，例如 6.1.1 小节例中的 `print(strname)`。
- 返回值：函数执行完的返回内容，用 `return` 开始。如没有返回值，则可以不写。例如 6.1.1 小节例中没有返回值。

对于一般类型的函数来讲，函数名和函数体是必须有的，函数的参数和返回值是可选的。

2. 文档字符串

在函数体中，常会在开始位置放置一个多行字符串，用来说明函数的功能及调用方法。这个字符串叫作函数的文档字符串（docstring），可以使用代码 `print(函数名.__doc__)` 将其输出。例如：

```
def getrecoder():                                #定义一个函数 getrecoder
    '''该函数是返回用户名字和年龄'''
    name = 'Gary'
    age = 32
    return name, age                             #返回 name 和 age 的值
print(getrecoder.__doc__)                        #返回文档字符串。输出：该函数是返回用户名字和年龄
```

例子中，在函数体的第一句就放置了一个文档字符串。该字符串用来描述函数。



提示：

在实际应用中，文档字符串主要用于描述函数的相关信息，以使用户更好地浏览和输出。建议养成在代码中添加文档字符串的好习惯。

6.1.3 函数的参数：形参与实参

形参，是从函数的角度来说的。6.1.2 小节中谈到的参数都是属于形参。

实参，是从调用的角度来说的。在调用时传入的参数就是实参。

6.1.1 小节例子中的调用代码：`hello("I love Python!")`，其中 `"I love Python!"` 就是传入 `hello` 函数中的实参，它是实际的参数值。而 `hello` 的参数 `strname` 就是形参。在函数执行时，在函数

hello 实现的功能是将参数 `strname` 打印出来。形参 `strname` 的值就是 "I love Python!", 于是屏幕就会输出 "I love Python! "。

6.1.4 函数的返回值

前面介绍过, 函数不需要返回值时可以什么都不做。需要有返回值时, 就要使用 `return` 语句将具体的值返回。使用 `return` 语句可以一次返回多个值。调用时, 可以定义多个变量来接收, 也可以使用一个元组来接收。例如:

```
def getrecoeder():                #定义一个函数 getrecoeder
    name = 'Gary'
    age = 32
    return name,age                #返回 name 和 age 的值

myname,myage = getrecoeder()      #在调用时, 使用与返回值对应的两个值来接收
print(myname,myage)               #将返回值打印出来, 输出: Gary 32

person = getrecoeder()            #在调用时, 使用与返回值对应的一个值来接收
print(person)                     #将返回值打印出来, 输出: ('Gary', 32)
```

有时可能只是需要用到返回值中的一个, 而将其他的忽略掉。这种情况下, 可以使用下划线 (`_`) 来接收对应返回值。例如:

```
personname,_ = getrecoeder()      #在调用时, 使用_来接收不需要的返回值
print(personname)
```

6.1.5 函数的属性

在 6.1.2 小节 “2. 文档字符串” 中讲到过函数的文档字符串, 它是存放在函数的 `__doc__` 属性中。这个属性是可更改的, 可以修改函数的 `__doc__` 属性值来为函数指定新的文档字符串。例如:

```
def getrecoeder():                #定义一个函数 getrecoeder
    '''该函数是返回用户名字和年龄'''
    name = 'Gary'
    age = 32
    return name,age                #返回 name 和 age 的值
print(getrecoeder.__doc__)        #返回文档字符串。输出: 该函数是返回用户名字和年龄
getrecoeder.__doc__ = "新文档字符串" #修改该函数__doc__属性值
print(getrecoeder.__doc__)        #打印文档字符串。输出: 新文档字符串
```

类似这样的功能，在定义函数的同时，还可以为函数添加自定义属性。每个函数都有个默认的字典__dict__，用于放置函数的属性。例如：

```
def recoder(strname,*,age):          #定义一个函数 recoder
    print ('姓名: ',strname,'年纪: ',age)    #函数的内容为一句代码，实现将指定内容输出

print(recoder.__dict__)              #函数 recoder 没有任何属性，__dict__为空。输出: {}
recoder.attr= "person"               #为函数 recoder 添加属性 attr, attr 的值为 person
print(recoder.__dict__)              #再次观察__dict__的值。输出: {'attr': 'person'}
print(recoder.attr)                 #可以直接将 attr 的值取出来。输出: person
```

直接在函数名后面加个点，就可以在后面直接添加属性；获取该属性时，也同样是在该函数名后面加个点，再跟上函数的具体属性名。这种方式可以在使用函数时传递更多的信息。

6.1.6 函数的本质

函数的本质是对象。可以被调用的函数，都会继承了可调用的方法 call，所以可以使用函数 callable 来检查某函数是否可以被调用。例如：

```
def recoder(strname,*,age):          #定义一个函数 recoder
    print ('姓名: ',strname,'年纪: ',age)    #函数的内容为一句代码，将指定内容输出

print( callable(recoder) )           #测试函数 recoder 是否可以被调用。输出: True
```

例子中，使用了 callable 函数判断 recoder 是否可以被调用。结果返回 True，表明函数 recoder 是可以被调用的。

6.1.7 函数的分类

函数有多种分类方法。

1. 从用户角度看

从用户角度看，函数可以分为内置函数、自定义函数。

- 内置函数：Python 内部自带的函数。
- 自定义函数：用户自己写的函数。

2. 从功能角度看

从功能上又可以分为生成器函数、一般函数。生成器函数见 6.8 节。

3. 从形式的角度来看

从形式的角度来看，Python 中的函数主要有五种形式：

- 普通的函数：使用 `def` 来定义。
- 匿名函数：使用 `lambda` 关键字来定义。
- 可迭代函数：属于一种特殊的内置函数。
- 递归函数：自己调用自己的函数。
- 偏函数：使用 `partial` 关键字来定义。

下面先介绍普通函数的使用方法。下文还会对匿名函数、可迭代函数、递归函数和偏函数单独介绍。

6.1.8 实例 18：打印两个心形图案

下面通过打印两个心形图案来演示函数的调用。

实例描述

在屏幕上输出两种心形图案：

一个心形图案是由 “I Love Python” 样式的文字组成，是实心。

另一个心形图案是由 “*” 字符组成，是空心。

第一种实现方法相对较简单，所用的是 5.7 节的知识。第二种实现方法相对复杂，需要编写函数的方式实现。

1. 输出实心的心形

仿照 5.7 节中的代码，直接编写一行语句即可。代码如下：

代码 6-1：打印心形图案

```
01 print('\n'.join([''.join(['I Love Python'[(x-y)%len('I Love Python')])if ((x*0.05)**2+(y*0.1)**2-1)**3-(x*0.05)**2*(y*0.1)**3<=0 else' ')for x in range(-32,32)])for y in range(17,-17,-1)])
```

代码使用了列表推导式将要打印的内容生成，并放到列表里。接着使用了两个 `join` 方法将字符串连接起来。这部分内容不是本例重点，具体的知识请参考 5.7 节。代码运行后，生成如图 6-1 所示心形。


```

thonILove      PythonILO
ePythonILovePytho   ovePythonILovePyth
vePythonILovePythonILovePythonILovePython
vePythonILovePythonILovePythonILovePythonIL
vePythonILovePythonILovePythonILovePythonILOv
ePythonILovePythonILovePythonILovePythonILOv
PythonILovePythonILovePythonILovePythonILOvP
ythonILovePythonILovePythonILovePythonILOvPy
thonILovePythonILovePythonILovePythonILOvPyth
honILovePythonILovePythonILovePythonILOvPyth
nILovePythonILovePythonILovePythonILOvPyth
LovePythonILovePythonILovePythonILOvPyth
ovePythonILovePythonILovePythonILOvPytho
PythonILovePythonILovePythonILOvPyth
thonILovePythonILovePythonILOvPyth
onILovePythonILovePythonILOvPyth
LovePythonILovePythonILOvPyth
ePythonILovePythonILOvPyth
thonILovePythonILOvPyth
ILovePythonILOv
ePythonILOv
hon
n

```

图 6-1 实心心形

图 6-1 中可以看到生成了一个实心心形，里面的内容是“ILovePython”。

2. 输出空心的心形

要实现空心心形需要两部分代码：一部分用来实现其中的空心部分，另一部分用来实现底部的实心部分。这里使用了正弦与正切函数合成的曲线来绘制。代码中分为 3 个函数：一个绘制心形的函数 `printstar`，两个条件判断函数 `comparefu1` 与 `comparefu2`。绘制心形函数会调用两个条件判断函数，根据不同的条件来控制形状。具体代码如下：

代码 6-1：打印心形图案（续）

```

02 import math                                #引入 math 库
03
04 def comparefu2(x,y):                        #定义绘制实心部分的条件函数
05     var3 = abs(y) - 0.65 * x * x
06     return var3 >=0.2
07
08 def comparefu1(x,y):                        #定义绘制空心部分的条件函数
09     H=3
10     var1 = math.sqrt(x * x + y * y) - H * math.sin(2 * math.atan(y / x))
11     var2 = math.sqrt(x * x + y * y) - H * math.sin(2 * math.atan(-y / x))
12     return abs( var1 )<= 1 or abs(var2)<= 1
13
14 def printstar(y,stepy,comparefun):          #定义绘制心形函数
15     while(y <= 0):
16         x = -4
17         while(x <= 4) :
18             if comparefun(x,y):
19                 print(" ",end=' ')

```


1. 列表方式定义参数与调用方式：fun(参数 1, 参数 2,...)

这是最常见的定义方式。一个函数可以定义任意个参数，参数间用逗号分隔。例如：

```
def recoder(strname,age):           #定义一个函数，名称为 recoder
    print ('姓名: ',strname,'年纪: ',age)  #函数的内容为一句代码，将指定内容输出
```

(1) 函数的调用：按照形参顺序依次传入

单独的函数是运行不起来的。需要对其调用才可以执行。调用时，直接使用函数名称，并且还要提供相同个数的实参。默认的情况下，实参顺序需要与形参一一对应。例如：

```
recoder ("Gary", "32")           # 调用函数，这时候屏幕会输出“姓名: Gary 年纪: 32”
```

上面代码中 recoder 函数有两个参数 strname、age。调用函数 recoder 时，也必须传入两个参数。在没有任何指定的情况下，这两个实参还需要与形参的顺序一一对应。

(2) 函数的调用：传入指定形参

在调用函数时，还可以直接为某个指定的形参赋值。在指定具体形参的情况下，传入的参数可以与形参的顺序无关。例如，上面的调用语句还可以写成这样：

```
recoder(age=32,strname=" Gary ") # 通过指定形参的方式调用函数，输出“姓名: Gary 年纪: 32”
```

这次传入实参顺序不一样了，并且 age 的参数类型也不是字符串了，而是一个整型。因为 Python 中的变量在定义时是不需要指定类型的。只有在调用时，系统才会根据传入的实参(32)定义函数(recoder)中的形参(age)的类型(int)。所以在该例子中，可以为一个函数的形参传入两个不同类型的实参。

传入指定形参的方式也是有限制的。如果在函数体里做了某个特殊类型的操作，那就必须传入这个类型的实参。例如，改写上面的 recoder 函数如下：

```
def recoder(strname,age):           #定义一个函数 recoder
    return age+1                     #将 age+1 返回
```

上面的代码中，在函数体里，将形参 age 进行“加 1”操作，并且使用 return 关键字返回该值。这很显然是将 age 当成了数值类型来处理。这时如果传入字符串就会报错，因为字符串不能与一个整型常量“1”相加。这种错误在函数调用时，会带来许多不必要的麻烦。

**提示:**

为防止函数体内对参数的操作与参数类型不一致,可以提前对参数类型进行检查。当传入错误类型时,被调函数能够及时发现。具体的方法见 6.2.2 小节的内容。

2. 星号方式定义参数与调用方式

在参数的列表中还可以直接使用星号(*),代表调用函数时,在星号的后面的参数都必须指定参数名称,例如:

| | |
|--|--|
| <code>def recoder(strname,*,age):</code> | <code>#定义一个函数 recoder, 要求形参 age 必须被指定</code> |
| <code>print ('姓名: ',strname,'年纪: ',age)</code> | <code>#函数的内容为一句代码, 实现将指定内容输出</code> |
| <code>recoder("Gary",age = 32)</code> | <code>#调用函数, 并指定形参 age</code> |
| <code>recoder("Gary",32)</code> | <code>#错误写法, 因为没有指定形参 age</code> |

例子中,函数 `recoder` 的形参使用了星号,星号后面为形参 `age`。这表明该函数被调用时 `age` 必须被指定。接下来又给出了两种调用方法:第一种指定了形参 `age`;第二种为错误方法,因为没有指定形参 `age`。

3. 带默认实参的列表方式,定义参数与调用方式: fun(参数 1,参数 2=值 2,...)

这种定义方式是对第一种定义的改进,为某些参数提供了默认值。

在调用时,被提供默认值的形参不需要有实参与其对应。没有传入实参的形参,会自动取默认值为其初始化。

例如:

| | |
|--|-------------------------------------|
| <code>def recoder(strname,age=32):</code> | <code>#定义一个函数 recoder</code> |
| <code>print ('姓名: ',strname,'年纪: ',age)</code> | <code>#函数的内容为一句代码, 实现将指定内容输出</code> |

调用时,传入一个或两个参数即可,例如:

| | |
|-------------------------------------|--|
| <code>recoder ("Gary", "32")</code> | <code># 调用函数, 传入两个参数。输出 “姓名: Gary 年纪: 32”</code> |
| <code>recoder ("Gary")</code> | <code># 调用函数, 传入一个参数。输出 “姓名: Gary 年纪: 32”</code> |

**提示:**

有默认值的形参,必须放在没有默认值的形参后面,否则会报错。例如下面的是错误的写法:

| | |
|---|---------------------|
| <code>def recoder(age=32,strname):</code> | <code>#错误的写法</code> |
|---|---------------------|

4. 通过元组或列表的解包参数的方式，定义参数与调用方式：fun(*参数)

这种定义方式只有一个形参，当被调用时，形参会被定义为一个元组。传入的实参都是这个元组类型的形参的元素。在函数体中，可以通过访问形参中的元素来获取传入的实参。

(1) 函数的调用：传入任意多的实参

这种定义方式在调用时，可以传入任意多的实参。例如：

```
def recoder (*person):                #定义一个函数，形参person的类型为元组
    print('姓名: ', person[0], '年纪: ', person[1]) #函数的内容为一句代码，实现将指定内容输出
```

调用时，传入两个参数，例如：

```
recoder ("Gary", "32")                # 调用函数，传入两个参数。输出“姓名: Gary 年纪: 32”
```

上面代码中，调用 `recoder` 传入了多个实参是可以的，但只传一个实参是有问题的。因为在函数 `recoder` 的函数体会获取形参的第二个元素（`person[1]`），所以，只传入一个实参相当于 `person` 只有一个元素，获取其第二个元素自然会失败。这是需要注意的地方。

另外，还有两点要注意的地方：

- 这种方式无法通过指定形参名称来传入实参，而且传入的实参顺序与形参内部元素的顺序必须一一对应；
- 因为接收参数的类型是元组，所以，用这种方式传值后不能对形参内容进行修改。

(2) 函数的调用：传入列表或元组

这里再介绍另一种调用方式，它可以将列表或元组当作实参传入。具体做法是，在列表或元组前加上一个星号。例如：

```
Mylist = ["Gary", "32"]                #定义一个 list
recoder (*Mylist)                      #调用函数，传入 list 作为实参。输出：姓名: Gary 年纪: 32
```

到这可以发现，函数 `recoder` 的形参是 `*person`，是将接收的参数当作元组，而调用时传入的 `*Mylist` 是一个列表的类型前面加个星号。

读者可以思考一下：如果把 `person` 前面的 `*` 与 `Mylist` 前面的 `*` 同时去掉，是不是也可以呢？它与加一个 `*` 的传递有什么区别呢？这部分内容将在 6.2.3 小节详细讲解。

5. 通过字典的解包参数方式，定义参数与调用方式：fun(**参数)

这是一种更为灵活的传参方式：传入的实参同时为其定义一个形参。这样在函数里就可以通过指定具体形参名称来获取实参了。这种方式是将形参当成一个字典类型变量来接收实参。这样传入的实参和对应的名字就可以放到这个字典里。形参为字典中元素的 **key**，实参为字典中元素的 **value**。取值时，直接通过字典里的 **key** 找到 **value**。例如：

```
def recoder (**person):          #定义一个函数，形参person的类型为字典
    print('姓名: ', person['name'],'年纪: ', person['age'])    #函数的内容为一句代码，实现将
指定内容输出
```

(1) 函数的调用：传入指定形参

调用时，传入实参的同时也指定了形参名称，例如：

```
recoder(age=32,name=" Gary ")    # 指定形参名称调用函数，输出：姓名：Gary 年纪：32
```

如果使用这种调用，就必须为形参指定名称，否则系统会报错误。例如：

```
recoder ("Gary", "32")          # 错误的写法
```

(2) 函数的调用：传入字典

调用时，还可以直接将字典传入。具体方法是，在传入的字典变量前加两个星号。例：

```
Mydic = {"name":"Gary", "age":"32"} #定义一个字典
recoder (**Mydic)                  #调用函数，传入list 作为实参。输出“姓名：Gary 年纪：32”
```

6. 总结：混合使用

最后介绍一下更为复杂的情况。当一个函数中的参数是通过多种方式定义时，应该如何对其调用。

(1) 字典和元组的解包参数，同时作为形参来接收实参

具体做法为：定义两个形参，第一个前有一个星号，用来接收实参并转为元组；第二个前有两个星号，用来接收实参并转为字典。例如：

```
def recoder(*person1,**person2):    #定义一个函数recoder，包括两个形参
    if len(person1)!=0:              #如果元组的形参接收到内容，就打印
        print ('姓名: ',person1[0],'年纪: ',person1[1])
    if len(person2)!=0:              #如果字典的形参接收到内容，就打印
        print ('姓名: ',person2["name"],'年纪: ',person2["age"])
```

调用时，可以为指定形参传值，也可以不指定形参直接传值。例如：

```
recoder("Gary",32)           #调用函数 recoder，传入不指定形参的实参，由 person1 接收
recoder(age=32,name="Gary")  #调用函数 recoder，传入指定形参的实参，由 person2 接收
```

还可以将指定形参的实参与不指定形参的实参，同时放入函数来调用。例如：

```
recoder("Gary",32,age=32,name="Gary")  #传入指定形参的实参与不指定形参的实参，person1、
person2 同时接收
```

上面这种写法必须是：不指定形参的实参在前，指定形参的实参在后，否则会报错。例如：

```
recoder(age=32,name="Gary","Gary",32)  #错误写法
```

(2) 字典或元组解包参数，与单个形参的混合使用

直接将字典或元组的解包参数与单个形参放在一起即可。但是，放置的先后顺序会影响到调用时的写法。

① 元组解包参数在前，单个形参在后时，需要如下的写法：

```
def recoder(*person1, ttt):      #定义一个函数 recoder，两个形参
    if len(person1)!=0:
        print('姓名:', person1[0], '年纪:', ttt)
recoder("Gary", ttt=32)          #调用时需要指定后面的单个形参，输出“姓名: Gary 年纪: 32”
```

元组解包参数在前，单个形参在后时，调用语句必须指定形参名称。

② 单个形参在前，元组解包参数在后时，需要如下的写法：

```
def recoder(ttt,*person1):      #定义一个函数 recoder
    if len(person1)!=0:
        print('姓名:', ttt, '年纪:', person1[0])
recoder("Gary",32)              #调用时不需要指定形参，输出“姓名: Gary 年纪: 32”
```

函数的单个形参在前，元组解包参数在后时，调用语句不需要形参名称。当然，传入实参时指定形参名称也是可以的。

(3) 字典解包参数、元组的解包参数、单个形参，三者一起使用。

当字典解包参数、元组的解包参数与单个形参放在一起时，必须保证字典的解包参数放在最后。例如：

```
def recoder(ttt,*person1,**arg):  #定义一个函数 recoder
    if len(person1)!=0:
        print('姓名:', ttt, '年纪:', person1[0])
```

```
recoder("Gary",32) #调用时不需要指定形参，输出“姓名：Gary 年纪：32”
```

当字典解包参数、元组的解包参数、单个形参放在一起，需注意三者的顺序，对于调用的规则，还是与前面（1）（2）点一致。

如果将第一个形参（`ttt`）与第二个形参（`*person1`）颠倒一下，也是可以的。例如：

```
def recoder(*person1, ttt,**arg): #定义一个函数 recoder
    if len(person1)!=0:
        print ('姓名: ',ttt,'年纪: ',person1[0])
recoder("Gary",ttt=32) #调用时不需要指定形参，输出“姓名：Gary 年纪：32”
```

按照前面（2）的规则，当元组的解包参数在单个形参前面时，单个形参需要被指定。所以，调用时第二个实参指定了形参 `ttt`。

如下的写法就是错误的：

```
def recoder(*person1, **arg, ttt): #错误，arg 没有在最后
    if len(person1)!=0:
        print ('姓名: ',ttt,'年纪: ',person1[0])
```

上例中字典的解包参数在中间，没有在最后，所以错误。这是个必须要注意的地方。即便形参中带有默认实参，也需要放到字典的解包参数前面。例如：

```
def recoder(*person1,ttt=9,**arg): #ttt 给定了一个默认值，但是它也得放在 arg 前面
    if len(person1)!=0:
        print ('姓名: ',ttt,'年纪: ',person1[0])
```

上例子中，函数 `recoder` 的形参 `ttt` 给定了一个默认值。一般情况下，这种带默认值的形参需要放在最后面（这里的先后指的是从左到右的顺序），但是有了 `arg` 的存在，带默认值的形参就需要放在 `arg` 前面、其他形参的后面。

那么在 Python 中，函数的实参与形参是如何传递的呢？下面就来看看具体规则。

6.2.2 在函数调用中检查参数

在函数调用中，默认是不会检查参数的类型。这种做法增加了代码的灵活度，但同时也增加了程序出错的概率。而在工业生产环境中应用的程序，更需要的是代码的健壮性。这时需要手动检查函数的参数，以提升代码的健壮性。具体的方法是，通过使用 `isinstance` 函数来对参数进行检查。

`isinstance` 函数的作用是检查变量的类型。具体定义如下：


```
isinstance(obj, class_or_tuple)
```

- 第一个参数 `obj`：传入待检查的具体对象。
- 第二个参数 `class_or_tuple`：待检查的具体类型，可以是类或者元组。

确切的说，该函数的功能是检查 `obj` 是否为 `class_or_tuple` 的一个实例。关于类和实例的内容可以在第 9 章学到。

在使用的过程中，直接在函数体的开始部分使用 `isinstance` 判断一下参数即可。`isinstance` 的第一参数传入带判断的形参；第二个参数就是合法的类型。例如：

```
def recoder(strname,age):                #定义一个函数 recoder
    if not isinstance(age, (int, str)):    #对参数类型进行检查，合法的类型为 int 和 str
        raise TypeError('bad operand type') #如果类型错误，使用 raise 函数进行报错
    print ('姓名: ',strname,'年纪: ',age)  #函数的内容为一句代码，实现将指定内容输出

recoder("Gary",age = 32.2)                #调用时传入了 age 为 float 类型
```

上面的代码中，`age` 允许传入的类型为整型和字符串类型，但是这里传入了浮点型，所以报错。这里使用了 `raise` 函数进行了报错提示并退出函数。关于 `raise` 类似的功能及介绍可以参考第 7 章内容。

6.2.3 函数调用中的参数传递及影响

在 Python 的内部机制里，函数调用时，实参传递到形参的过程中有两种情况——传值和传引用。

- 对于不可变对象，函数调用时，是传值。意味着函数体里可以使用实参的值，但不能改变实参。
- 对于可变对象，函数调用时，是传引用。意味着在函数体里还可以对实参进行修改。

具体情况举例如下。

1. 不可变对象

不可变对象的传值是指：当传入的实参指向一个常量，或是一个元组等不允许修改的对象时，就把该对象的值传递给形参。例如：

```
def fun(arg):                #定义一个函数
    arg = 5                  #在函数体里通过为形参赋值的方式改变形参
x = 1                        #定义一个变量 x，让它等于 1
```

```

fun(x)           #输出将 x 当作实参传入函数 fun
print(x)         #打印 x，因为 x 的值（值为 1）为不可变对象，所以其值不变。输出： 1

```

上面这段代码把 x 作为参数传递给函数，这时 x 和 arg 都指向内存中值为 1 的对象。

在函数体中，执行了 arg = 5。由于 int 对象不可改变，于是创建一个新的 int 对象（值为 5）并且令 arg 指向它。而 x 仍然指向原来的值为 1 的 int 对象，所以函数没有改变 x 变量。

2. 可变对象

可变对象的传递引用是指：当传入的实参指向一个 list，或是一个字典等允许修改的对象时，就把该对象的值传递给形参。例如：

```

def fun(arg):      #定义一个函数，形参为一个列表
    arg.append(3)   #在函数体里，通过为形参添加一个元素的方式改变形参
x = [1, 2]         #定义一个列表 x
fun(x)            #将 x 传入函数
print(x)          #执行完函数 fun，列表 x 发生了变化，输出：[1, 2, 3]

```

这段代码同样把 x 传递给函数 fun。x 和 arg 都会指向同一个 list 类型的对象。

函数体中使用列表的 append 方法在末尾添加了一个元素。因为列表对象是可以改变的，所以列表对象的内容发生了改变。由于 x 和 arg 指向的是同一个 list 对象，所以变量 x 的内容也发生了改变。

3. 综合分析

在 6.2.1 小节中的“6. 通过元组或列表的解包参数的方式：fun(*参数)”部分，抛出了一个问题，使用解包参数与列表传递有什么不同。看下面的例子：

```

Mylist = ["Gary", "32"]
def recoder (person):      #定义一个函数，形参为 person，未指定类型
    person[0] = 'sss'      #修改 person 中元素的值
    print('姓名: ', person[0], '年纪: ', person[1]) #将指定内容输出
recoder (Mylist)           #调用 recoder，输出“姓名: sss 年纪: 32”
print(Mylist)              #将 Mylist 打印，输出：['sss', '32']

```

这个例子是一个可变对象的传值，Mylist 的值在调用完 recoder 函数后发生了变化。

接下来演示解包参数传值的情况。将函数 recoder 的参数变为* person。调用函数 recoder 时传入*Mylist。代码如下：

```

Mylist = ["Gary", "32"]

```

```
def recoder (*person):
    person[0] = 'sss'                ##修改 person 中元素的值
    print('姓名: ', person[0], '年纪: ', person[1]) #将指定内容输出

recoder (*Mylist) # 调用函数, 传入*Mylist。会报错误, 提示函数 recoder 中不可以修改 person 的值
print(Mylist)
```

上例中, 使用解包参数的方式进行值传递, 并且在 `recoder` 函数中对 `person` 进行了修改。这是错误的写法, 因为 `person` 是元组类型, 不支持修改。所以, 在调用时会报错误。

这两个例子可以说明解包参数和直接列表传值的区别。使用解包参数的函数, 不能对参数修改; 而使用列表传值的函数, 是可以对参数进行修改的。

6.3 匿名函数与可迭代函数

匿名函数一般适用于单行代码函数。

匿名函数的作用是, 把某一简单功能的代码封装起来, 让整体代码更规整一些。一般只调用一次就不再需要, 所以名字也省略了, 于是变成了匿名函数。

6.3.1 匿名函数与可迭代函数的介绍

匿名函数是以关键字 `lambda` 开始的, 它的写法如下:

```
Lambda 参数 1, 参数 2...: 表达式
```

上面的写法中, 表达式的内容只能是一句话的函数, 而且不能存在 `return` 关键字。

例如:

```
r = lambda x,y:x*y                #定义一个匿名函数实现 x 与 y 相乘
print( r(2,3))                   #传入 2 和 3, 并把它们打印出来。输出: 6
```

`lambda` 表达式可以在任何地方使用, 它相当于一个可以传入参数的单一表达式。例如, 在一般函数里这样使用:

```
def sum_fun(n):                    #定义个函数
    return lambda x: x+n          #返回一个匿名函数

f = sum_fun(15)                   #得到一个匿名函数, 函数体为 x+15
print(f(5))                      #向匿名函数里传入 5。输出: 20
```

在上面的例子中，在函数 `sum_fun` 里返回了匿名函数，并为其指定了一个被加数。在调用过程中，只需要放入另一个加数，便实现了两个数相加。

匿名函数本质与函数是没什么两样的。在实际应用中，匿名函数常会与可迭代函数配合使用。可迭代函数就是一种有循环迭代功能的内置函数，包括 `reduce`、`map` 和 `filter` 等。在每个可迭代函数中都需要指定一个处理函数，习惯上使用匿名函数作为处理函数，当然使用普通函数也是有效的。下面就来一一介绍。

6.3.2 匿名函数与 `reduce` 函数的组合应用

`reduce` 函数的功能是按照参数 `sequence` 中的元素顺序，来依次调用函数 `function`，并且每次调用都会向其传入两个参数：一个是 `sequence` 序列中的当前元素，另一个是 `sequence` 序列中上一个元素在函数 `function` 中的返回值。其定义如下：

```
reduce(function, sequence, [initial])
```

- **function:** 要回调的函数。
- **sequence:** 一个“序列”类型的数据。
- 第三个参数可选，是一个初始值。

`reduce` 函数本质上也可以算作一个内嵌循环的函数。

`reduce` 函数与匿名函数的结合使用，能够以更为简洁的代码实现较复杂的循环计算功能。例如，下面使用 `reduce` 函数与匿名函数的结合写法，来实现求 1~100 的和：

```
from functools import reduce                                #导入 reduce 函数
print(reduce (lambda x,y:x + y,range(1,101) ) )           #第一个参数是个匿名函数,实现两个数相加,
输出: 5050
```

函数里，通过匿名函数实现了两个数的求和，然后使用 `range` 函数得到一个 1~100 的列表。依次取出列表里的值，将它们加在一起。

`reduce` 函数一般用于归并性任务。

6.3.3 匿名函数与 `map` 函数的组合应用

类似 `reduce` 函数，匿名函数还可以与 `map` 函数组合。

`map` 函数的功能是：将参数 `sequence` 内部的元素作为参数，并按照 `sequence` 序列的顺序，依次调用回调函数 `function`。具体定义：

```
map(function, sequence[, sequence, ...])
```

- **function**: 要回调的函数。
- **sequence**: 一个或多个“序列”类型的数据。

该函数返回值为一个 `map` 对象。在使用时，得用 `list` 或 `tuple` 等函数进行转化。

1. 使用 map 函数处理一个序列数据

当 `map` 后面直接跟一个序列数据时，直接将该序列数据中的元素作为参数，依次传入前面的函数。例如：

```
t = map(lambda x: x ** 2,[1, 2, 3, 4, 5]) #使用map函数,对列表[1,2,3,4,5]的元素求平方值。返回值赋给t
print(list(t)) #将t转成列表类型,并打印。输出: [1, 4, 9, 16, 25]
```

例子中，`map` 函数会将传入的列表`[1, 2, 3, 4, 5]`中的每个元素传入匿名函数里，进行平方运算，得出的值会放入新的 `map` 对象中，最后将整个 `map` 对象赋给变量 `t`。通过 `list` 函数对 `t` 进行类型转换，生成新的列表。在新的列表里，每个元素都是原来列表元素平方后的结果。

2. 使用 map 函数处理多个序列数据

当 `map` 后面直接跟多个序列数据时，处理函数的参数个数要与序列数据的个数相同。

运行时，`map` 内部依次提取每个序列数据中的元素，一起放到所提供的处理函数中，直到循环遍历完最短的那个序列。例如：

```
t = map(lambda x,y: x+y,[1, 2, 3, 4, 5],[1, 2, 4, 5])#遍历最短的列表[1,2,4,5],实现两个列表的元素相加
print(list(t)) #将t转成列表类型,并打印。输出: [2, 4, 7, 9]
```

该例子是对两个序列中的元素依次求和。新生成的列表中的元素分别是，两个原序列中对应位置的元素相加。

第一个序列长度是 5，第二个序列长度是 4。两个序列长度不相等，循环会以最小长度对所有序列进行提取。于是，新生成的列表中也只有 4 个元素。

`map` 函数一般用于映射性任务。

6.3.4 匿名函数与 filter 函数的组合应用

`filter` 函数的功能是对指定序列进行过滤。

1. filter 函数的定义

`filter` 函数有两个输入参数：一个是 `filter` 的处理函数，另一个是待处理的序列对象。在运行时，`filter` 函数会把序列对象中的元素依次放到 `filter` 的处理函数中。如果返回 `True`，就留下，反之就舍去。其定义如下：

```
filter(function or None, sequence)
```

- **function:** 为 `filter` 的处理函数，在 `filter` 的内部以回调的方式被调用。返回布尔型。意味着某元素是否要留下。
- **sequence:** 是一个或多个“序列”类型的数据。

2. filter 函数的使用

`filter` 函数的返回值是一个 `filter` 类型，需要将其转成列表或元组等序列才可以使用。例如：

```
t=filter(lambda x:x%2==0, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])    #筛选出一个列表中为偶数的元素
print(list(t))                                                #转成列表，并打印。输出[2, 4, 6, 8, 10]
```

例子中，通过 `filter` 来过滤数组中偶数的元素。

如果 `filter` 的处理函数为 `None`，则返回结果和 `sequence` 参数相同。例如：

```
t=filter(None, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) #过滤一个列表中为偶数的元素
print(list(t))                                #转成列表，并打印。输出[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

当 `filter` 的处理函数为 `None` 时，返回的元素与原序列一样。从功能上来说，没有什么意义；从代码框架的角度来说，会更有扩展性。当处理函数的输入是变量时，则可以把 `filter` 函数的调用部分代码固定下来。为处理函数变量赋值不同的过滤函数，以实现不同的处理功能。

`filter` 函数一般用于过滤性任务。

6.3.5 可迭代函数的返回值

每个可迭代函数返回的值都属于一个生成器对象（在 6.8 节还会详细介绍）。

生成器对象与迭代器对象（迭代器对象在前面 5.8 节有介绍）用法类似，生成器对象会有一个 `__next__` 方法。调用该方法可以取到内部的各个值。最终 `__next__` 将通过 `StopIteration` 异常来停止迭代。例如：

```
t=filter(lambda x:x%4==0, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])    #过滤列表中被 4 整除的元素，返回元素 4、8
print(t.__next__())                                           #使用__next__方法获取元素，输出：4
print(t.__next__())                                           #使用__next__方法获取元素，输出：8
print(t.__next__())                                           #已经没有元素，返回 StopIteration
```

例子中，通过 `filter` 得出的结果 `t` 中只有两个元素，通过调用 `__next__` 方法可以取得下一个元素。如没有元素，则返回 `StopIteration` 异常。因为 `filter` 遍历元素的原理与 `for` 循环语句的处理很类似，所以也可以使用 `for` 循环来将 `t` 对象内容依次取出。例如上面代码可以改写成：

```
t=filter(lambda x:x%4==0, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])    #过滤列表中被 4 整除的元素，返回元素 4、8
for a in t:                                                     #使用循环取出 t 中的所有元素
    print(a)                                                    #打印出来，输出：4 8
```

可以看到，可迭代函数的返回结果可以直接转换成列表或元组，也可以使用循环来取得。这里只是拿 `filter` 举例，所有可迭代函数都可以这样使用。

生成器对象与迭代器对象的主要区别在于：生成器对象只能迭代一次，而迭代器对象可以迭代多次。所以，对可迭代函数的结果只能取一次，无法再次提取。这一点一定要注意。例如：

```
t=filter(lambda x:x%4==0, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])    #过滤列表中被 4 整除的元素，返回元素 4、8
for a in t:                                                     #使用循环取出 t 中元素
    print(a)                                                    #打印出来，输出：4 8
print(list(t))                                                  #已经取过一次，所以这次就没了。输出[]
```

上面代码中，使用了一个 `for` 循环将 `filter` 的返回值打印出，接着想再次用 `list` 函数将 `filter` 的返回值转换成列表，并同时打印出来。由于生成器对象只能取一次，所以转化的结果为空，打印的内容也是空。

6.4 偏函数

偏函数是对原始函数的二次封装，它是属于寄生在原始函数上的函数。偏函数可以理解为重新定义一个函数，向原始函数添加默认参数。有点像面向对象中的父类与子类的关系。

1. 偏函数的定义

偏函数的关键字是 `partial`，其定义如下：

```
partial(func, *args, **keywords)
```

- **func**: 要封装的原函数;
- 第二个参数为一个元组或列表的解包参数 (见 6.2.1 节中 6): 代表传入原函数的默认值 (可以不指定参数名);
- 第三个参数为一个字典的解包参数 (见 6.2.1 节中 7): 代表传入原函数的默认值 (指定参数名)。

其中, 第二个参数与第三个参数的作用是一样的, 都是代表传入原函数的参数。偏函数的作用是, 为其原函数指定一些默认的参数。调用该偏函数, 就相当于调用了原函数, 同时将默认的参数传入。在使用 `partial` 前, 必须引入 `functools` 模块。

2. 偏函数的使用

下面通过例子说明:

```
from functools import partial
def recoder(strname, age):           # 定义一个函数 recoder
    print('姓名: ', strname, '年纪: ', age)   # 函数的内容为一句代码, 实现将指定内容输出

Garyfun = partial(recoder, strname="Gary") # 定义一个偏函数
Garyfun(age = 32)                       # 调用偏函数, 传入 age = 32。输出 "姓名: Gary 年纪: 32"
```

偏函数的本质是, 将函数式编程、默认参数和冗余参数结合在一起。通过偏函数传入的参数调用关系, 与正常函数的参数调用关系是一致的。

偏函数通过将任意数量 (顺序) 的参数, 转化为另一个带有剩余参数的函数对象, 从而实现了截取函数功能 (偏向) 的效果。在实际应用中, 可以使用一个原函数, 然后将其封装多个偏函数, 在调用函数时全部调用偏函数。这样的代码可读性提升了很多。

6.5 递归函数

递归函数, 就是自己调用自己的函数。要在递归函数体里写出跳出函数的条件, 否则会产生无穷的调用, 最终导致栈溢出。因为, 所有的函数调用都是一个压栈的过程, 而系统为每个进程分配的栈空间是有限的。一旦发生无穷调用时, 系统会不停往栈里写入函数地址, 直到把栈写满, 最后程序崩溃。

Python 语言中, 对函数的递归调用没有做特别的优化处理。这会导致一个问题: 一旦循环调用自己的次数足够多, 也会将栈写满, 从而导致程序崩溃。为了程序的健壮性, 作者不建议

在程序中使用递归函数。凡是用递归函数可以解决的，用循环函数都可以解决。所以，作者建议多用循环函数来解决，而不是用递归函数。

6.6 eval 与 exec 函数

`eval` 与 `exec` 函数属于 Python 的内置函数。由于这两个函数的特殊性，有必要单独说明一下。更多的内置函数介绍可以参见本书附件部分。

`eval` 与 `exec` 都可以执行一个字符串形式的 Python 代码（代码以字符串的形式提供），相当于一个 Python 的解释器。在使用 Python 开发服务端程序时，这两个函数应用得非常广泛。比如，客户端向服务端发送一段字符串代码，服务端无需关心具体的内容，直接跳过 `eval` 或 `exec` 来执行。这样的设计会使服务端与客户端的耦合度更低，系统更易扩展。

6.6.1 eval 与 exec 的区别

先通过一个例子展示 `eval` 与 `exec` 的用法，代码如下：

```
exec("print(\"I love Python \")") # exec 里面传入一句代码。输出: I love Python
eval("print(\"I love Python \")") # eval 里面传入一句代码。输出: I love Python
```

二者不同的是：`eval` 执行完要返回结果，而 `exec` 执行完不返回结果。例如下面的代码：

```
a = 1
exec("a = 2")           #相当于直接执行 a=2
print(a)                #输出 a 的值为 2
a = exec("2+3")         #相当于直接执行 2+3，但是并没有返回值，a 应为 None
print(a)                #输出 a 的值为 None
a = eval('2+3')         #执行 2+3，并把结果返回给 a
print(a)                #输出 a 的值为 5
```

可以看出，`exec` 中最适合放置运行后没有结果的语句，而 `eval` 中适合放置有结果返回的语句。如果 `eval` 里放置一个没有结果返回的语句会怎样呢？例如下面代码：

```
a = eval("a = 2")
```

这时会报错。提示 `eval` 中不识别等号语法。

6.6.2 eval 与 exec 的定义

对 `exec` 和 `eval` 的定义有三个参数（`expression`, `globals=None`, `locals=None`），具体含义如下：

- **expression:** 这个参数是一个字符串，代表要执行的语句。该语句受后面两个字典类型参数 **globals** 和 **locals** 限制，只有在 **globals** 字典和 **locals** 字典的作用域内的函数和变量才能被执行。
- **globals:** 这个参数管控的是一个全局的命名空间，即 **expression** 可以使用全局的命名空间中的函数。如果只是提供了 **globals** 这个参数，而没有提供自定义的 **__builtins__**，则系统会将当前环境中的 **__builtins__** 复制到自己提供的 **globals** 中，然后才会进行计算；如果连 **globals** 这个参数都没有被提供，则使用 Python 的全局命名空间。
- **locals:** 这个参数管控的是一个局部的命名空间，和 **globals** 类似。当它和 **globals** 中有重复或冲突时，以 **locals** 的为准。如果 **locals** 没有被提供，则默认为 **globals**。

1. exec 与 eval 中的 globals 参数作用

下面通过例子来演示参数 **globals** 作用域的作用，读者可以观察它是何时将 **__builtins__** 复制 **globals** 字典中去的。代码如下：

```
dic={}           #定义一个字典
dic['b']=3       #在 dic 中加一条元素，key 为 b
print(dic.keys()) #先将 dic 的 key 打印出来，有一个元素 b。输出: dict_keys(['b'])
exec("a = 4", dic) #在 exec 执行的语句后面跟一个作用域 dic
print(dic.keys()) #exec 后,dic 的 key 多了一个,输出: dict_keys(['a', '__builtins__', 'b'])
```

上面的代码，是在作用域 **dic** 下执行了一句 **a=4** 的代码。可以看出，在 **exec** 之前 **dic** 中的 **key** 只有一个 **b**。执行完 **exec**，系统在 **dic** 中生成了两个新的 **key**：**a** 和 **__builtins__**。其中，**a** 为执行语句生成的变量，系统将其放到指定的作用域字典里；**__builtins__** 是系统加入的内置 **key**。

2. 了解内建模块 __builtins__

__builtins__ 是 Python 的内建模块，例如我们平时使用的 **int**、**str**、**abs** 等都在这个模块中。

通过下面的代码可以查看 **__builtins__** 所对应的 **value**：

```
print(dic["__builtins__"])           #接上面代码，将 __builtins__ 的 value 取出
```

输出如下：

```
{'chr': <built-in function chr>, 'RuntimeWarning': <class 'RuntimeWarning'>,
'IsADirectoryError': <class 'IsADirectoryError'>,
.....
'ArithmeticError': <class 'ArithmeticError'>, 'UserWarning': <class 'UserWarning'>,
'globals': <built-in function globals>}
```

3. 演示 exec 与 eval 中作用域

为了更好地理解作用域的功能，看下面的代码：

```
dic={}                                #定义一个字典
a = 2
exec("a = 4", dic)                   #在 exec 执行的语句后面跟一个作用域 dic
print(a)                             #打印 a 的值。发现 a 没有变化。输出 2
print(dic['a'])                       #其实执行的语句所生成的变量是在 dic 中。输出：4
```

上面的代码表明 exec 执行的语句中，赋值的变量 a 并不是全局的变量 a，而是新生成了一个 a，存放在 dic 中。

下面再来看看 eval 函数的使用。代码如下：

```
dic={}                                #定义一个字典
dic['a'] = 3                          #往字典里加个键值对 a: 3
dic['b'] = 4                          #往字典里加个键值对 b: 4
result = eval('a+b',dic)              #让 dic 内部的 a 与 b 对应的值相加，结果赋给 result
print(result)                         #将结果打印出来，输出：7
```

在上面的代码中，eval 中的语句是计算 a 加 b，系统会自动上 dic 中找 a 和 b 对应的 value。如果里面没有 a 或者 b，程序会报错。

4. exec 与 eval 中的 locals 参数作用

再来看看指定 locals 的情况下：

```
a=10
b=20
c=30
g={'a':6, 'b':8}                      #定义一个字典
t={'b':100, 'c':10}                   #定义一个字典
print(eval('a+b+c',g,t))              #定义一个字典 116
```

6.6.3 exec 和 eval 的使用经验

使用 exec 和 eval 时一定要记住：里面的第一个参数是字符串，而字符串的内容一定要是可执行的代码。

下面以 eval 为例，用代码演示常犯的错误：

```
s="hello"
```

```
print(eval(s))          # 错
```

这个例子出错的地方在于，字符串的内容是 `hello`，而 `hello` 并不是可执行的代码（除非定义了一个变量叫作 `hello`）。

1. `exec` 与 `eval` 的可接受参数

如果要将字符串 `hello` 通过 `print` 函数打印出来，可以写成如下的样子：

```
s="hello"
print(eval('s'))        #对
```

这种写法是要 `eval` 执行 `"hello"` 这句代码。这个 `hello` 是有引号的，在代码中，代表字符串的意思，所以可以执行。同理，也可以写成这样：

```
s='hello'
print(eval(s))          #s是个字符串，字符串的内容是带引号的hello
                        #输出hello
```

这种写法的意思是 `s` 是个字符串，并且其内容是个带引号的 `hello`。所以直接将 `s` 放入到函数 `eval` 中也可以执行。

2. `repr` 函数在 `exec` 与 `eval` 函数中的作用

还可以不去改变原有字符串 `s` 的写法，直接使用 `repr` 函数来进行转化，也可以得到同样的效果。例如：

```
s="hello"
print(eval(repr(s)))    # 使用函数 repr 进行转化，输出hello
```

虽然函数 `eval` 与 `str` 的返回值都是字符串。但是使用 `str` 函数对 `s` 进行转化，程序同样会报错。这是值得注意的地方。例如：

```
s="hello"
print(eval(str(s)))     # 错
```

为什么会有这个区别呢？

同样对带字符串 `s` 的转化，使用 `repr` 与 `str` 得到的结果是有差别的，直接将二者的结果打印出来就可以很明显地看出不同。见下面代码：

```
s="hello"
print(repr(s))          # 输出: 'hello'
print(str(s))           # 输出: hello
```

可见使用 `repr` 返回的内容，输出后会在两边多一个单引号。这一功能与前面的“1. `exec` 与 `eval` 的可接受参数”部分完全一致，这也是其内部的真正原理。

**注意：**

在编写代码时，一般会使用 `repr` 函数来生成动态的字符串，再传入到 `eval` 或 `exec` 函数内，实现动态执行代码的功能。`repr` 函数的更多介绍可以参考 9.6.1 小节的“注意”部分。

6.6.4 `eval` 与 `exec` 的扩展知识

如果读者以后接触到 TensorFlow 框架，就会发现 TensorFlow 中的静态图就是类似这个原理实现的。TensorFlow 中先将张量定义在一个静态图里。这就相当于 6.6.2 小节例中，将键值对添加到字典里一样；TensorFlow 中通过 `session` 和张量的 `eval` 函数来进行具体值的运算（这部分知识在作者的另一本书《深度学习之 TensorFlow：入门、原理与进阶实战》里有详细介绍）。就相当于 6.6.3 小节例中，使用 `eval` 函数进行具体值的运算一样。

另外，在使用 `eval` 或是 `exec` 来处理请求代码时也要额外小心。函数 `eval` 和 `exec` 常常会被黑客利用，成为可以执行系统级命令的入口点，进而来攻击网站。解决方法是，通过设置其命名空间里的可执行函数，来限制 `eval` 和 `exec` 的执行范围。

6.7 实例 19：批量测试转化函数（实现“组合对象”与“字符串”的相互转化）

本例子将演示 `exec` 与 `eval` 的用法。利用 `exec` 进行批量函数调用，利用 `eval` 进行列表、元组、字典与字符串的相互转化。

实例描述

编写两个功能函数：一个将“组合对象”转化为“字符串”，另一个将“字符串”转化为“组合对象”。编写若干个单元测试案例，并对案例进行批量测试。

该案例属于单元测试中的一个常见案例。单元测试是软件工程中的重要步骤，是指程序员要对自己所完成的每个模块或函数单元，进行功能性测试。这种开发模式可以保证软件的基础质量，减少在后期系统集成时的错误。

本实例分三步骤实现：

- (1) 实现两个功能函数。
- (2) 编写测试用例。
- (3) 进行批量调用。

6.7.1 编写两个功能函数

定义函数 `stringtoOther` 与 `OthertoString`。

- `stringtoOther`: 通过调用 `eval` 函数，将组合对象的类型转化为字符串。
- `OthertoString`: 通过调用 `str` 函数，将字符串转化为组合类型。

代码如下：

代码 6-2: 使用 `exec` 函数批量调用多个函数

```
01 def stringtoOther(a):                #组合对象类型转化为 string
02     b = eval(a)
03     return b
04
05 def OthertoString(b):                #string 转化为组合对象类型
06     a = str(b)
07     return a
```

使用 `eval` 函数将字符串转化为组合对象，是一个常用的技巧。读者可以将此用法记住，在自己的项目中使用。

6.7.2 编写单元测试用例

在单元测试步骤，会使用组合对象的 3 种情况分别举例，每个案例用一个函数封装起来。具体代码如下：

代码 6-2: 使用 `exec` 批量调用多个函数（续）

```
08 def fun1():                          #字符串与列表相互转换
09     a = "[[1,2], [3,4], [5,6], [7,8], [9,0]]"
10     b = stringtoOther(a)
11     print(b)
12     print(OthertoString(b))
13
14 #####
15 def fun2():                          #字符串与字典相互转换
```

```

16     a = "{1: 'a', 2: 'b'}"
17     b = stringtoOther(a)
18     print(b)
19     print(OthertoString(b))
20
21 #####
22 def fun3():                                     #字符串与元组相互转换
23     a = "([1,2], [3,4], [5,6], [7,8], (9,0))"
24     b = stringtoOther(a)
25     print(b)
26     print(OthertoString(b))

```

上面代码中，每一个单元测试的案例函数都使用了相同的逻辑：先定义一个具有组合对象内容的字符串，然后其转为真正的组合对象，再将其转回成字符串。

6.7.3 批量运行单元测试用例

使用 for 循环，构造出单元测试函数名，并用 exec 函数进行批量调用将其执行。代码如下：

代码 6-2：使用 exec 批量调用多个函数（续）

```

27 if __name__ == '__main__':
28     s=""
29     for i in range(1,4):
30         s+='fun'+str(i)+'()\n'
31     exec(s)

```

代码运行后，输出如下结果：

```

[[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]
{1: 'a', 2: 'b'}
{1: 'a', 2: 'b'}
([1, 2], [3, 4], [5, 6], [7, 8], (9, 0))
([1, 2], [3, 4], [5, 6], [7, 8], (9, 0))

```

从结果中可以看到。功能函数分别进行了列表与字符串（输出结果的头两行）、字典与字符串（输出结果的中间两行）、元组与字符串（输出结果的后两行）的相互转化，单元测试成功，表明功能函数的逻辑正确。

6.8 生成器函数

在 6.3.5 小节中提到过“生成器对象”的概念，它与迭代器对象的主要区别——只能迭代一次。这里再详细说明一下。

6.8.1 生成器与迭代器的区别

虽然生成器对象的使用方法与迭代器对象类似，但是内部的原理却完全不同。

- 迭代器是所有的内容都在内存里，使用 `next` 函数来依次往下遍历。
- 生成器不会把内容放到内存里，每次调用 `next` 函数时，返回的都是本次计算出来的那个元素，用完之后立刻销毁。

所以，当整个序列占用内存特别大时，使用生成器对象会节约内存；当系统的运算资源不足时，使用迭代器对象会节约运算资源。二者在应用上的区别属于“时间优先”与“空间优先”之间的区别。

下面再来介绍一下生成器函数。

6.8.2 生成器函数

生成器函数（Generator）是一种特殊的函数，是用来创建生成器对象的工具。生成器函数使用 `yield` 语句返回，返回值是一个生成器对象。例如：

```
def Reverse(data):                #定义一个函数实现字符串反转
    for idx in range(len(data)-1, -1, -1):
        yield data[idx]          #使用yield返回具体的一个元素

for c in Reverse('Python'):       #使用for循环来迭代Reverse返回的生成器对象
    print(c, end=' ')            # 输出: n o h t y P
```

上面的代码中，定义了函数 `Reverse`，其功能是将一个字符串倒序，并返回生成器对象。在调用中，使用 `for` 循环对生成器对象进行遍历，并打印出来。在 `for` 循环的内部实现里，每次循环生成器都会返回一个元素，等全部循环结束后，生成器也随之结束，不会在内存里有残留。

6.8.3 生成器表达式

生成器还有一种简单的写法，就是用生成器表达式。它适合一句代码的语句。生成器表达

式与 5.6 节中讲过的“for 循环列表推导式”写法很像，也是使用 for 循环来实现的。不同的是，“for 循环列表推导式”外层是方括号，而生成器表达式外层是圆括号。例如：

```
mylist = [x*x for x in range(3)]    #使用列表推导式为一个新定义的列表赋值
for i in mylist:                    #使用 for 循环将其打印出来
    print(i)                        #输出：0 1 4
```

例子中，使用了“for 循环列表推导式”对 0~2 的数字计算平方值，并以列表的形式输出。当所有语句执行完 mylist 的值还是存在的。如果换成生成器表达式，则写法如下：

```
myGen = ( x*x for x in range(3) )    #使用生成器表达式返回一个生成器对象
for i in myGen:                      #使用 for 循环将其打印出来
    print(i)                        #输出：0 1 4
```

上面是个生成器表达式的写法。这个例子执行后输出也是 0、1、4。不同的是，生成器 myGen 的值在打印输出之后就不在了。

6.9 变量的作用域

Python 为变量分配了 4 个作用域，代表程序中出现同名变量的有效范围。

- L：本地作用域，被当前函数包括。
- E：上一层结构中 def 或 lambda 的本地作用域（其实就是函数嵌套的情况）。
- G：全局作用域，不被任何函数包括。
- B：内置作用域，是 Python 内部的命名空间。

这几个作用域的优先级由高到低排列，依次为 L→E→G→B，俗称 LEGB 原则。意思是：如果代码里用到了某个变量（例如：a），系统内部会按照 LEGB 的顺序去不同的作用域里面找变量 a，并在第一个能够找到变量名的地方停下来；如果在这 4 个作用域中都没找到，Python 会报错。

6.9.1 作用域介绍

前三个作用域（LEG）是分布在用户代码中的，在自己的工程代码中可以看到。下面通过代码演示：

```
var = 1                                #在全局作用域 G 中定义个变量 var，值为 1
def fun1():                            #定义一个函数 fun1，其内部的函数体都属于作用域 E
    def fun2():                        #在函数 fun1 中定义一个函数 fun2，其内部的函数体都属于作用域 L
```

```

    var = 3          #在 L 中定义一个变量 var，值为 3
    print(var)       #输出 var 的值
    var = 2          #在 E 中定义变量 var，值为 2
    fun2()           #在函数 fun1 中调用了 fun2

fun1()              #调用函数 fun1，最终会进入 fun2，fun2 中的 var 生效。输出：3

```

这个例子中，`fun1` 中嵌入了一个函数 `fun2`。`fun2` 函数中调用了其函数体内自己定义的变量 `var`。输出的结果为 3。

如果将 `fun2` 函数体内的变量 `var` 去掉，代码如下：

```

var = 1              #在全局作用域 G 中定义个变量 var，值为 1
def fun1():          #定义一个函数 fun1，其内部的函数体都属于作用域 E
    def fun2():       #在函数 fun1 中定义一个函数 fun2，其内部的函数体都属于作用域 L
        print(var)    #输出 var 的值
        var = 2       #在 E 中定义变量 var，值为 2
        fun2()         #在函数 fun1 中调用了 fun2

```

调用 `fun1`，屏幕会输出 2。因为 `fun2` 中没有了变量 `var`，根据 LEGB 原则会去 E 作用域中找，在 E 作用域中有一个 `var` 值为 2，所以就输出了 2。

如果接着再将 `fun1` 函数中的 `var` 定义去掉，代码如下：

```

var = 1              #在全局作用域 G 中定义个变量 var，值为 1
def fun1():          #定义一个函数 fun1，其内部的函数体都属于作用域 E
    def fun2():       #在函数 fun1 中定义一个函数 fun2，其内部的函数体都属于作用域 L
        print(var)    #输出 var 的值
        fun2()         #在函数 fun1 中调用了 fun2

```

再调用 `fun1`，屏幕会输出 1。系统会根据 LEGB 原则，逐层去找 `var`。最终在 G 作用域下找到，此时 G 作用域下的 `var` 会生效。

第四个作用域 B 属于 Python 中的内置作用域，可以通过如下代码查看：

```

import builtins
dir(builtins)

```

输出结果如下：

```

['ArithmeticError',
.....
'tuple',
'type',
'vars',

```

```
'zip']
```

6.9.2 global 语句

如果在本地作用域 L 或嵌套作用域 E 下，想对全局作用域 G 的变量进行赋值等操作，可以使用 `global` 语句。

例如：

```
a=6                #全局变量 a 为 6
def func():        #定义一个函数
    global a       #获得全局变量 a
    a=5            #对 a 赋值 5
func()            #调用函数 func
print(a)          #将 a 的值打印出来，输出 5
```

例子中，函数 `func` 修改了全局变量 `a`，所以最终输出的值为 5。如果不加 `global` 这句，将会输出 6。例如：

```
a=6                #全局变量 a 为 6
def func():        #定义一个函数
    a=5            #对 a 赋值 5
func()            #调用函数 func
print(a)          #将 a 的值打印出来，输出 6
```

这个例子中，函数 `func` 为 `a` 赋值，`a` 会被当作一个函数 `func` 的本地作用域下的变量处理，所以并没有对全局变量 `a` 进行赋值。打印全局变量 `a` 时，最终输出的还是 6。

6.9.3 nonlocal 语句

`global` 语句是可以引用全局的优先级变量。类似 `global` 语句，`nonlocal` 可以引用比其优先级低的作用域下的变量。如果在本地作用域 L 或嵌套作用域 E 下，想对全局作用域 G 的变量赋值等操作，可以使用 `global` 语句；而使用 `nonlocal` 语句，则会在本地作用域以外按照优先级的顺序逐级去寻找声明的变量，并引用该变量。例如：

```
a=6                #全局变量 a 为 6
def func():        #定义一个函数
    a=7            #定义嵌套作用域下的变量 a
    def nested():  #内嵌函数 nested
        nonlocal a #使用 nonlocal 关键字，引用外层的 a
        a+=1       #对 a 进行加 1 操作，改变 a 的值
    nested()       #调用函数 nested
```

```

print("本地: ",a)           #将本地变量打印。输出“本地: 8”
func()                      #调用函数 func
print("全局: ",a)          #将 a 的值打印出来，输出“全局: 6”

```

例子中，函数 `func` 内使用了内嵌函数 `nested`。在 `nested` 体内对 `func` 的变量进行了修改。所以最终输出的值中全局变量没变，是 6；`func` 的本地变量变了，是 8。

6.10 工厂函数

工厂函数某种程度上实现了面向对象思想的编程方法（面向对象思想将在第 9 章有介绍），它可以让代码更有层次。在复杂程序架构中，利用工厂函数的思想来设计架构，会使得代码更有扩展性。

6.10.1 普通工厂函数的实现

首先利用作用域的知识来实现一个偏函数的功能。例如 6.4 节中的偏函数例子，还可以改写如下：

```

def recoder(strname,age):    #定义一个函数 recoder
    print('姓名: ',strname,'年纪: ',age) #函数的内容为一句代码，实现将指定内容输出

def Garyfun(age):           #实现了偏函数的功能
    strname = 'Gary'        #定义了本地作用域下的变量
    return recoder(strname,age) #直接将固定的变量 strname 传入
Garyfun(age = 32)           #调用生成器函数，传入 age = 32。输出“姓名: Gary 年纪: 32”

```

其实上述这个写法就已经实现了工厂函数。为了更深入理解，接着上面的代码再加一个函数。如下：

```

def Annafun(age):           #再定义一个工厂函数 Annafun
    strname = 'Anna'        #定义了本地作用域下的变量
    return recoder(strname,age) #直接将固定的变量 strname 传入
Annafun(age = 37)           #调用生成器函数，传入 age = 37。输出“姓名: Anna 年纪: 37”

```

上面两段代码中有两个工厂函数，分别对 `recoder` 进行封装：一个是 `Garyfun`，放置的是默认名字 `Gary`；另一个是 `Annafun`，放置的默认名字是 `Anna`。它们有同样的属性，都可以传入年龄，并将名字和年龄一起输出。

例子中，基于这种编程思想的实现，需要编写不同的封装函数，每一个函数里都指定了一个 `strname` 变量。分别对名字 `Anna` 与 `Gary` 实现了两个函数的封装。如果要支持的 `strname` 名

字有 n 个，是不是需要写 n 个封装函数与之对应呢？这显然是不可取的方法。

下面介绍一种简化的方式来解决这个问题，使用闭合函数（closure）。

6.10.2 闭合函数（closure）

闭合函数又叫闭包函数，本质上与普通工厂函数的编程思想一致，是普通工厂函数的更优形式，由自由变量与嵌套函数组成。

1. 闭合函数的实现

闭合函数实现方法是：将名字作为自由变量，将原有的 `recoder` 函数作为嵌套函数，通过一次函数的封装即可实现前面的功能。代码如下：

```
def wrapperfun(strname):                # 闭合函数，strname 为自由变量
    def recoder(age):                  # 定义一个嵌套函数 recoder
        print('姓名: ', strname, '年纪: ', age)    # 函数的内容为一句代码，实现将指定内容输出
    return recoder                      # 返回 recoder 函数

fun = wrapperfun('Anna')               # 自由变量设为 Anna
fun(37)                                # 为 age 赋值，输出“姓名: Anna 年纪: 37”
fun2 = wrapperfun('Gary')              # 自由变量设为 Gary
fun2(32)                                # 为 age 赋值，输出“姓名: Gary 年纪: 32”
```

闭合函数 `wrapperfun` 中实现了一个嵌入函数 `recoder`，并将嵌入函数 `recoder` 返回。嵌入函数 `recoder` 中调用了外部变量 `strname`，这个 `strname` 是由 `wrapperfun` 被调用时传入的参数。当 `wrapperfun` 被调用后，返回自身嵌入 `recoder` 的同时，又将自身的参数 `strname` 与内嵌 `recoder` 绑定起来，赋值给了 `fun`。`fun` 就等同一个 `strname` 被初始化后的 `recoder` 函数。此时的 `fun` 可以理解为一个 `recoder` 的闭合函数，自由变量 `strname` 存在于该闭合函数之内。

2. 闭合函数的 `__closure__` 属性

闭合函数会比普通的函数多一个属性——`__closure__`，该属性会记录着自由变量的参数对象地址。当闭合函数被调用时，系统就会根据该地址找到自由变量，完成整体的函数调用。

接着上面的代码，演示查看 `__closure__` 属性，代码如下：

```
print(fun.__closure__)                # 输出(<cell at 0x000000000B49E4C8: str object at
0x000000000C497030>,)
```

可以看到，显示的内容是一个字符串对象，这个对象就是 fun 中自由变量 strname 的初始值。`__closure__` 属性的类型是一个元组，表明闭合函数可以支持多个自由变量的形式。

6.10.3 装饰器 (decorator)

装饰器是 Python 语言中专门为软件工程服务的编程方法。在软件工程中，一个项目的多个版本间迭代要尽量遵循“开发封闭”的原则。即，对于已经实现的功能代码不允许被修改，但可以被扩展。

1. 装饰器的实现

装饰器的主要作用就是在扩展原有功能基础上，最大化地使用已有代码。也可以理解成：在不改变原有代码实现的基础上，添加新的实现功能。

装饰器的实现方法是：在原有的函数外面再包装一层函数，使新函数在返回原有函数之前实现一些其他的功能。

例如，可以修改 6.10.2 小节中的 `wrapperfun` 函数，为其增加参数校验的功能。修改后的代码如下：

```
def checkParams(fn):                                #装饰器函数，参数是要被装饰的函数。相当于闭合函数
    def wrapper(strname):                          #定义一个检查参数的函数
        if isinstance(strname, (str)):             #判断是否是字符串类型
            return fn(strname)                     #如是，则调用 fn(strname) 返回计算结果
        print("variable strname is not a string tpye") #如果参数不符合条件，则打印警告，
    然后退出
    return wrapper                                  #将装饰后的函数返回

def wrapperfun(strname):                            #闭合函数，strname 为自由变量
    def recoder(age):                               #定义一个嵌套函数 recoder
        print ('姓名: ',strname,'年纪: ',age)      #函数的内容为一句代码，实现将指定内容输出
    return recoder                                  #返回 recoder 函数

wrapperfun2 = checkParams (wrapperfun ) #对 wrapperfun 进行装饰，即，将自由变量设为
wrapperfun 函数
fun = wrapperfun2 ('anna')                        # wrapperfun2 为带有参数检查的闭合函数
fun(37)                                             #为 age 赋值，输出“姓名: Anna 年纪: 37”
fun = wrapperfun2 (37) #当输入参数不合法时，输出: variable strname is not a string type
```

这段代码中，添加了一个 `checkParams` 函数。`checkParams` 是装饰器函数，返回值为内部定

义的 `wrapper` 函数。为了实现对原有函数 `wrapperfun` 的参数检查，将 `wrapper` 函数的参数与 `wrapperfun` 参数保持一致。若被检查的参数合法，再调用原有函数，并将参数透传进去；若被检查的参数不合法，则打印警告。

在使用时，直接将函数 `wapperfun` 传入到 `checkParams` 中去，来完成对原函数 `wrapperfun` 的装饰，并得到带有参数检查的闭合函数 `wrapperfun2`。于是在最后一行代码中传入值为 37 而不是字符串，系统则打印警告。

2. 装饰器的本质与用途

装饰器本质是一个闭合函数，该闭合函数的自由变量是一个函数。它的存在可以使代码的重用性与扩展性大大加强。

可以想象这样一种场景：假设前面 6.10.2 小节中的例子是某个系统低版本的部分代码。在使用过程中，发现在调用 `wrapperfun` 函数时，偶尔会输入一个数字或非字符串类型导致程序出错。

面对这种情况，就可以在新的版本中对 `wrapperfun` 进行装饰，使其具有参数检查的功能。实际使用时，可以将本节例子代码中的 `wrapperfun2` 全部变成 `wrapperfun` [装饰函数的语句就会变成：`wrapperfun = checkParams (wrapperfun)`]，这样得到的 `wapperfun` 就具有了参数检查的功能，代码的改动量也很小。

在实际情况中，装饰器的应用场景非常广泛。除了为函数添加参数检查外，还可以为函数添加调试信息、日志等。

6.10.4 @修饰符

6.10.3 小节中的装饰器函数，还有另外一种写法——使用@修饰符。

@修饰符的作用是，在定义原函数时就可以为其指定修饰器函数。这等同于将 6.10.3 小节中的代码 `wrapperfun = checkParams(wrapperfun)` 移到了 `wrapperfun` 的定义部分。

这样做的好处是：使修饰器与被修饰函数的关系更加明显，也使得需要修饰的函数在第一时间得到修饰，降低了编码出错的可能性。

@修饰符的语法是：在@后面添加修饰器函数，同时在其下一行添加被修饰函数的定义。将 6.10.3 小节的例子使用@修饰符的方法改写，代码如下：

```
def checkParams(fn):#装饰器函数，参数是要被装饰的函数。相当于闭合函数
```

```

def wrapper(strname):
    # 定义一个检查参数的函数
    if isinstance(strname, (str)):
        # 判断是否是字符串类型
        return fn(strname)
        # 如果是，则调用 fn(strname) 返回计算结果
    print("variable strname is not a string tpye")
    # 如果参数不符合条件，则打印警告，
    # 然后退出
    return
    # 将装饰后的函数返回
return wrapper

@checkParams
def wrapperfun(strname):
    # 使用@修饰符来实现对 wrapperfun 的修饰
    # 闭合函数，strname 为自由变量
    def recoder(age):
        # 定义一个嵌套函数 recoder
        print('姓名: ', strname, '年纪: ', age)
        # 函数的内容为一句代码，实现将指定内容输出
    return recoder
    # 返回 recoder 函数

fun = wrapperfun('Anna')
# wrapperfun 为带有参数检查的闭合函数
fun(37)
# 为 age 赋值，输出“姓名: Anna 年纪: 37”
fun = wrapperfun(37)
# 当输入参数不合法时，输出: variable strname is not a string type

```

使用一个@符号，就可以把修饰器与被修饰函数的关系简洁地体现出来。这也是编写代码中较常用的写法。

6.10.5 更高级的装饰器

了解完装饰器的原理之后，再来学习 Python 中更高级的装饰器。这些高级装饰器在真正编程场景中很常用，因为它可以灵活地适应很多情况。

1. 能够接收任何参数的通用参数装饰器

在 6.10.3 小节的例子中，装饰器 checkParams 的内部实现了一个 wrapper 函数。wrapper 函数的参数必须与被装饰函数参数一样，这样才能够实现对被装饰函数的参数检查，并将输入参数透传给被装饰函数。

为了在定义装饰器时，解耦内部 wrapper 的参数对于被装饰函数参数的强依赖关系，可以使用能够适应任何参数的通用参数装饰器。

通用参数装饰器的实现很简单，利用了 6.2.1 小节中“参数的定义”知识。即，在 wrapper 函数的参数中，使用字典和元组的解包参数来作为形参。这样得到的装饰器便可以适用于各种不同参数的函数。例如，将 6.10.4 小节的代码改写成通用参数装饰器的形式（只改变 checkParams 函数即可），如下：


```

def checkParams(fn):
    def wrapper(*arg, **kwargs):
        if isinstance(arg[0], (str)):
            return fn(*arg, **kwargs)
        print("variable strname is not a string tpye")
    return wrapper

```

然后退出

#定义通用参数装饰器函数
#使用字典和元组的解包参数来作形参
#判断第一个参数是否是字符串类型
#如满足条件，则将参数透传给原函数，并返回
#如果参数不符合条件，则打印警告，
#将装饰后的函数返回

用上面代码来替换 6.10.4 小节中的实例代码前 7 行，整个程序还可以正常运行。但是装饰器 `checkParams` 却变得更为灵活，它不仅仅只适应于对 `wrapperfun` 的装饰。同样适用于与 `wrapperfun` 参数不同的函数。只需要对第一个参数做字符串类型检查的函数，都可以用 `checkParams` 来装饰。这就是通用参数装饰器的便捷之处。

2. 可接收参数的通用装饰器

通用参数装饰器，解决了装饰器参数与被装饰函数间的强关联关系。为了让装饰器有更大的通用性，还可以通过在装饰器的外部传入参数的方式，来告诉装饰器当时使用的外部情况。这样，装饰器内部就可以通过传进来的外部变量来选择不同的执行分支，从而可以适应更多的调用场景。

可接收参数的通用装饰器，需要在原有的通用参数装饰器外面再封装一层函数，这个函数的参数就是用来接收外部变量的。例如：

```

def isadmin(userid):
    def checkParams(fn):
        def wrapper(*arg, **kwargs):
            if userid != 'admin':
                print('Operation is prohibited as you are not admin! ')
                return
            if isinstance(arg[0], (str)):
                return fn(*arg, **kwargs)
            print("variable strname is not a string tpye")
        return wrapper
    return checkParams

@ isadmin (userid='admin')
def wrapperfun(strname):
    def recoder(age):

```

#可以接收参数的装饰器函数
#用通用参数来装饰器函数
#定义一个检查参数的函数
#对外部调用环境进行判断，如不是 admin，则直接返回
#判断是否是字符串类型
#如满足条件，则将参数透传给原函数，并返回
#如果参数不符合条件，则打印警告，然后退出
#将装饰后的函数返回
#在 admin 模块中，传入 admin 身份到装饰器
#闭合函数，strname 为自由变量
#定义一个嵌套函数 recoder

```

    print ('姓名: ',strname,'年纪: ',age)    #函数的内容为一句代码，实现将指定内容输出
    return recoder                          #返回 recoder 函数

@ isadmin (userid='user')                  #在 user 模块中，传入 user 身份到装饰器
def wrapperfun2(strname):                 #闭合函数，strname 为自由变量
    def recoder(age):                     #定义一个嵌套函数 recoder
        print ('姓名: ',strname,'年纪: ',age)    #函数的内容为一句代码，实现将指定内容输出
        return recoder                   #返回 recoder 函数

fun = wrapperfun ('anna')                  # wrapperfun 为带有参数检查的闭合函数
fun(37)                                    #为 age 赋值，输出 “姓名: Anna 年纪: 37”
fun = wrapperfun2(37) #身份不对，输出: Operation is prohibited as you are not admin!
```

假设外部情况有两个模块：一个是 `admin`，另一个是 `user`。在上面的代码中，在通用参数装饰器 `checkParams` 外部添加一层封装，变为 `isadmin` 装饰器。该装饰器的作用是检查调用者的身份，只允许身份为 `admin` 的用户调用。在 `admin` 中实现了函数 `wrapperfun`，在 `user` 中实现了函数 `wrapperfun2`，两个函数都用 `isadmin` 来装饰。在实际调用时，`wrapperfun` 可以正常使用，但是 `wrapperfun2` 在调用时就会显示“Operation is prohibited as you are not admin!”信息。原因是：`isadmin` 装饰器中对传入的外部参数进行了判断，只有 `admin` 才会正常运行。

下面两段代码实现了同样的功能：一个使用了 `@` 注解的方法定义装饰器；另一个使用了普通的方法定义装饰器。具体如下：

(1) `@` 注解后面跟着可接收参数的装饰器的定义代码：

```

@ isadmin (userid='user')                  #在 user 模块中，传入 user 身份到装饰器
def wrapperfun2(strname):                 #闭合函数，strname 为自由变量
    def recoder(age):                     #定义一个嵌套函数 recoder
        print ('姓名: ',strname,'年纪: ',age)    #函数的内容为一句代码，实现将指定内容输出
        return recoder                   #返回 recoder 函数
```

(2) 普通装饰器的定义代码：

```

def wrapperfun2(strname):                 #闭合函数，strname 为自由变量
    def recoder(age):                     #定义一个嵌套函数 recoder
        print ('姓名: ',strname,'年纪: ',age)    #函数的内容为一句代码，实现将指定内容输出
        return recoder                   #返回 recoder 函数
```

装饰器函数定义好后，可以使用如下语句进行调用：

```
wrapperfun2 = isadmin (userid='user')(wrapperfun2)
```

上面这句代码执行时，系统会先执行 `isadmin (userid='user')` 返回一个装饰器 `checkParams` 函

数，再调用 `checkParams` 函数，并传入参数 `wrapperfun2`，返回 `wrapper` 函数。所以这时执行下列代码：

```
print(wrapperfun2.__name__)           #输出：wrapper
```

打印出来的是 `wrapper`。表明，最终会将 `wrapper` 函数返回，并赋值给 `wrapperfun2`。这反映了个客观的现象：装饰器在装饰函数时，改变了函数本身的名称。为了避免这个现象，见下面“3. 装饰器返回函数的名称修复”中的内容。

3. 装饰器返回函数的名称修复

当函数被装饰完后，对函数的名字属性再赋一次值，将函数的名称恢复过来。这样就可以避免出现装饰完后函数名字变化的现象。例如，将 `isadmin` 函数改成如下：

```
def isadmin(userid):                #可以接收参数的装饰器函数
    def checkParams(fn):            #通用参数装饰器函数
        def wrapper(*arg, **kwargs): #定义一个检查参数的函数
            if userid != 'admin':    #对外部调用环境进行判断，如不是，则 admin 直接返回
                print('Operation is prohibited as you are not admin! ')
                return
            if isinstance(arg[0], (str)): #判断是否是字符串类型
                return fn(*arg, **kwargs) #如满足条件，则将参数传递给原函数，并返回
            print("variable strname is not a string tpye") #如果参数不符合条件，则打印警告，然后退出
        return
        wrapper.__name__ = fn.__name__ #将函数名称属性恢复
        return wrapper                #将装饰后的函数返回
    return checkParams
```

上面代码中的倒数第三行，是新加的语句。该语句的意思是：将被装饰的函数名称赋给 `wrapper` 函数。这样，装饰器返回的 `wrapper` 函数名称，就与被装饰的函数名称一致了。

另外，Python 中还提供了一个内置的装饰器函数 `functools.wraps`，它的作用是将被装饰的函数名称还原并赋值给装饰后的返回函数。不过在使用时，需要先引入 `functools` 模块。例如：

```
import functools
def isadmin(userid):                #可以接收参数的装饰器函数
    def checkParams(fn):            #用通用参数来装饰器函数
        @functools.wraps(fn)        #内置的装饰器，用于恢复函数名称
        def wrapper(*arg, **kwargs): #定义一个检查参数的函数
            if userid != 'admin':    #对外部调用环境进行判断，如不是，则 admin 直接返回
                print('Operation is prohibited as you are not admin! ')
            return fn(*arg, **kwargs)
```

```

        return
    if isinstance(arg[0], (str)) :#判断是否是字符串类型
        return fn(*arg, **kwargs) #如满足条件，则将参数透传给原函数，并返回
    print("variable strname is not a string tpye") #如果参数不符合条件，则打印
警告，然后退出
    return
    return wrapper #将装饰后的函数返回
return checkParams

```

上面代码的第1行，引入了 `functools` 模块。紧接着，在第4行使用了@符号对 `wrapper` 进行装饰，这样 `wrapper` 的函数名称就会变成与传入的 `fn` 参数一样的函数名称。保证了装饰之后的函数与原函数的名称一致。

4. 组合装饰——多个装饰器联合使用

在软件工程学中，会非常注重代码的可复用性。装饰器的使用，可以使代码拥有更好的可复用性。在实际编码中，可以将具有不同功能的装饰器合起来用，以实现组合装饰的效果。

实现组合装饰时，仅需要将不同的装饰器使用@符号一行一行的堆叠起来即可。例如：

```

def checkParams(fn): #通用参数装饰器，用于检查参数
    def wrapper(*arg, **kwargs): #使用字典和元组的解包参数来作为形参
        if isinstance(arg[0], (str)) : #判断第一个参数是否是字符串类型
            return fn(*arg, **kwargs) #如满足条件，则将参数透传给原函数，并返回
        print("variable strname is not a string tpye") #如果参数不符合条件，则打印警告，
然后退出
    return wrapper #将装饰后的函数返回

def logging(userid): #可接收参数的通用装饰器，用于打印日志
    def checkParams(fn): #装饰器函数，参数是要被装饰的函数。相当于闭合函数
        def wrapper(*arg, **kwargs): #使用字典和元组的解包参数来作为形参
            print(userid,end=':') #将调用者的身份打印出来
            return fn(*arg, **kwargs) #将参数透传给原函数，并返回
        return wrapper #将装饰后的函数返回
    return checkParams

@logging(userid = 'admin') #多行@符号，实现组合装饰
@checkParams
def wrapperfun(strname): #被装饰函数
    def recoder(age): #定义一个嵌套函数 recoder
        print ('姓名: ',strname,'年纪: ',age) #函数的内容为一句代码，实现将指定内容输出
    return recoder #返回 recoder 函数

```

```
fun = wrapperfun ('anna')           # wrapperfun 为带有参数检查的闭合函数
fun(37)                             #打印的结果带有调用身份, 输出“admin:姓名: anna 年纪: 37”
```

上面代码中，实现了两个装饰器——`checkParams` 与 `logging`。`checkParams` 用于参数检查；`logging` 用于输出调用者的身份信息。

在函数 `wrapperfun` 定义的前行，使用了带@符号的装饰器 `checkParams` 与 `logging` 对其封装。在装饰器 `logging` 被调用时，传入了调用身份 `admin`。这样，在执行最后一行的代码时，就会输出的 `admin` 信息。

使用装饰器，可以将代码按照要实现功能的主次逐层分开（例如：核心功能、安全检查、日志信息等），使代码更有条理。将核心功能与参数检查、外围交互等功能逐层分开的思想，也大大降低了代码的复杂度。这就是面向切面的编程思想（Aspect Oriented Program, AOP）。

5. 多装饰器的调用顺序

上面“4. 组合装饰——多个装饰器联合使用”的例子中，使用了两个装饰器——`checkParams` 与 `logging`。这两个装饰器在实际运行中的顺序是什么样的呢？这里就来剖析一下。

多装饰器的载入顺序是从下往上的。在调用时，执行的函数顺序是从上往下的。怎么理解这句话呢？来看下面这段代码：

```
def logging(fn):                      #logging 函数
    print ('in logging')
    def wrapper_logging(*args, **kwargs):
        print ('in wrapper_logging')
        return fn(*args, **kwargs)
    return wrapper_logging

def checkParams(fn):                  # checkParams 函数
    print ('in checkParams')
    def wrapper_checkParams(*args, **kwargs):
        print ('in wrapper_checkParams')
        return fn(*args, **kwargs)
    return wrapper_checkParams

@logging
@checkParams
def wrapperfun(strname):              #被装饰函数
    print ('姓名: ',strname)          #函数的内容为一句代码: 将指定内容输出
```

上面代码中，每个函数里都加上一句打印的语句，用于演示其内部的调用关系。整个代码运行后，会输出如下结果：

```
in checkParams
in logging
```

这时，代码中根本没有调用的语句，但是同样会打印出内容。这表明：装饰器在整个代码载入时就开始执行了，而且顺序是按照@的顺序，从下往上加载装饰器函数（结果中先输出了下面的 `checkParams` 函数，后输出了上面的 `logging` 函数）。接下来再执行一次调用语句，如下：

```
wrapperfun("Anna")           #对装饰函数进行调用
```

这句代码运行后有下列输出：

```
in wrapper_logging
in wrapper_checkParams
姓名: Anna
```

这次的输出中，依照@的顺序，上面的 `wrapper_logging` 被先执行，下面的 `wrapper_checkParams` 被后执行。整体过程相当于一个压栈弹栈的过程，即，载入时下执行的装饰函数其返回的函数在最后被调用。这就是多装饰器的调用顺序。

6.10.6 解决“同作用域下默认参数被覆盖”问题

在使用工厂函数时，常会配合循环来批量生成多个函数。这里有个特例值得注意：如果通过循环来生成工厂函数，在循环过程中，该循环体作用域下的默认参数值会被循环值所覆盖。这种情况会导致，所有在这个循环中产生的函数都将会有相同的默认值。例如：

```
def recoder(strname,age):           #定义一个函数 recoder
    print ('姓名: ',strname,'年纪: ',age)   #函数的内容为一句代码，将指定内容输出

def makerecoders():                 #该函数的作用是，批量生成工厂函数
    acts=[]
    for i in ["Gary","Anna"]:       #通过循环将列表中的元素作为默认值，依次生成工厂函数
        acts.append(lambda age:recoder(i,age)) #将每个生成的函数放到列表里
    return acts                     #将生成的批量函数返回

for a in ( makerecoders()):          调用函数批量生成工厂函数，并使用 for 遍历每个函数
    a(age = 32)                     #将每个函数取出，依次次调用，输出如下
    # 姓名:  Anna 年纪:  32
```

```
# 姓名: Anna 年纪: 32
```

上面的例子中，想通过 for 循环遍历列表来批量生成工厂函数。其中，列表里的值为工厂函数的默认值。但在实际调用过程中发现，只有列表中最后的一个元素的值起到了默认值的作用，前面的元素都没有生效。这是由于，在循环时后面的变量覆盖了前面的变量。

为了避免这种情况的发生，在循环生成工厂函数的过程中，就不能将默认值放到作用域空间来存储，必须要将默认值当作参数传入到原函数 recoder 中。例如：

```
def recoder(strname,age):                #定义一个函数 recoder
    print ('姓名: ',strname,'年纪: ',age) #函数的内容为一句代码，实现将指定内容输出

def makerecoders():                      #该函数的作用是，批量生成工厂函数
    acts=[]
    for i in ["Gary","Anna"]:            #使用循环，将列表中的元素作为默认值，依次生成工厂函数
        acts.append(lambda age,i=i:recoder(i,age)) #将循环值 i 作为参数也传入到匿名函数里
    return acts                           #将生成的批量函数返回

for a in ( makerecoders()):               #调用函数批量生成工厂函数，并用 for 遍历每个函数
    a(age = 32)                           #将每个函数取出，一次调用。输出如下。
                                           # 姓名: Gary 年纪: 32
                                           # 姓名: Anna 年纪: 32
```

这次在函数 makerecoders 中的 for 循环内，将循环值 i 也作为参数传入了匿名函数里。避免了本地作用域下的变量在循环中会被覆盖的问题，实现了正确的功能。输出的两个函数：一个默认值为 Gary，另一个默认值为 Anna。

有关装饰器的更多例子还可以参见本书 7.6 节。

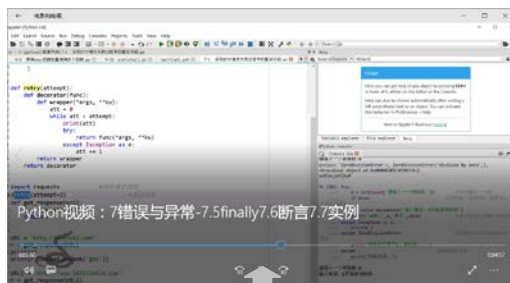
第 7 章

错误与异常——调教出听话的程序

开发人员在编写程序时，难免会遇到错误。有的是由于编写人员的疏忽所造成的语法错误；有的是因为程序内部具有隐含的逻辑问题而造成的数据错误；也有程序运行时与系统的规则冲突造成的系统错误……

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解

具体代码操作演示



Python视频：
7错误与异常
-7.1错误分类
7.2异常的基...



Python视频：
7错误与异常
-7.3常见异常
7.4创建异常...



Python视频：
7错误与异常
-7.5finally7.6
断言7.7实例...

本章共有 3 段教学视频，总时长为 23 min 左右。包含下内容：

- 讲述了错误的分类和异常的基本语法。
- 讲述了常用的异常以及如何创建异常。
- 讲述了异常的最终处理（清理动作）、判定条件的正确性（断言）和一个实例。例子中讲解了如 HTTP 请求失败，实现“重试”的功能。

7.1 错误的分类

编写程序过程中遇到的错误都分为两类：语法错误与运行时错误。

7.1.1 语法错误

语法错误，也就是在解析时所出现错误。当代码不符合 Python 语法规则时，在解析过程中会报 `SyntaxError`。报错的同时会显示出错的那一行，并且用小箭头指出最早探测到错误的位置。例如：

```
print 'hello'           #在 Python 3 中，这种写法是不允许的，'hello'外面必须有括号
```

上面是错误写法。因为在 Python 3 中，函数 `print` 被调用时，没有找到后面的括号，所以在运行时，就会报如下错误：

```
print 'hello'
File "<iPython-input-1-bf8e230352b8>", line 1
    print 'hello'
        ^
SyntaxError: Missing parentheses in call to 'print'
```

这种错误因为编写人员疏忽导致的，代码与基本的代码规则违背。它属于真正意义上的错误，是 Python 程序不能容忍的。程序执行过程中，发现有这种错误便立刻停止。

7.1.2 运行时错误

运行时错误，即语句或表达式在语法上都是正确的，但在运行时发生了错误。例如：

```
a = 1/0                # 让 1 除以 0，结果赋值给 a
```

上面这句代码的意思是“用 1 除以 0，并赋值给 `a`”。因为 0 作被除数是没有意义的，所以运行后会产生如下错误：

```
Traceback (most recent call last):  
  
  File "<iPython-input-2-82f326788aeb>", line 1, in <module>  
    a= 1/0  
  
ZeroDivisionError: division by zero
```

上面显示的内容中：前两段是错误的位置；最后一句是出错的类型。在 Python 中，会把这种运行时产生错误的情况叫做异常（Exceptions）。这种异常情况还有很多。例如：

```
a=3+t          #将 3 与 t 的和赋值给 a
```

因为上面代码中的 `t` 没有定义，所以会提示如下异常：

```
Traceback (most recent call last):  
  
  File "<iPython-input-3-01cf05da9aff>", line 1, in <module>  
    a=3+t  
  
NameError: name 't' is not defined
```

这种异常就是属于 `NameError` 类型的异常。例子中的 `NameError` 与前面的 `ZeroDivisionError` 都属于内置的异常名称。Python 中内建了许多类似 `NameError` 的全局标识符，用来规范标准异常类型的命名。

当一个程序发生异常时，代表该程序在执行时出现了非正常的情况，无法再执行下去。默认情况下，程序是要终止的。为了避免程序退出，可以使用捕获异常的方式获取这个异常的名称，再通过其他的逻辑代码让程序继续运行。这种根据异常做出的逻辑处理叫作异常处理。

开发者可以使用异常处理全面地控制自己的程序。异常处理不仅仅能够管理正常的流程运行，还能够在程序出错时对程序进行必要的处理。大大提高了程序的健壮性和人机交互的友好性。

7.2 异常的基本语法

Python 语法会把异常当作一个对象，通过 `try/except` 语句来捕捉该异常对象。`try/except` 语句后面都会跟着对应的代码块。系统会在 `try` 对应的代码块中内置检测错误的代码。当检测到错误时，就会进入 `except` 代码块，来执行相应的逻辑处理，以决定是否继续运行。

1. 异常的定义

异常处理的语法定义如下：

```
try:
<语句>          #运行别的代码
except <名字>:
<语句>          #如果在 try 部份引发了 'name' 异常
except <名字>, <数据>:
<语句>          #如果引发了 'name' 异常，获得附加的数据
else:
<语句>          #如果没有异常发生，则执行该分支语句
```

2. 异常的使用举例

一个 try 语句可以有多条 except 语句，用以指定不同的异常，但至多只有一个会被执行。例如：

```
try:
    x = int(input('请输入一个被除数:'))    #等待输入一个数
    print('30 除以', x, '等于', 30/x)      #输出 30 除以输入数字
except ValueError:                         #捕获 ValueError 异常
    print('输入了无效的整数。重新输入……')
except ZeroDivisionError:                  #捕获 ZeroDivisionError 异常
    print('被除数不等于 0，重新输入……')
except :                                  #捕获其他异常
    print('其他异常……')
```

上面这段代码中，共有三个分支处理 try 抛出的异常。

(1) 程序运行之后，当输入 a（非数字）时，将抛出 ValueError 异常，程序进入第一分支。输出“输入了无效的整数。重新输入……”。

(2) 当输入为 0 时，将抛出 ZeroDivisionError 异常，程序进入第二分支。输出“被除数不等于 0，重新输入……”。

(3) 假如运行过程中产生的异常既不等于 ValueError，又不等于 ZeroDivisionError，则执行第三分支，输出“其他异常……”。

7.2.1 同时处理多个异常

except 关键字还可以同时接收多个异常。具体的写法是在 except 后面加个括号，将要接收

的异常当作参数传入。例如：

```
try:
    x = int(input('请输入一个被除数:'))    #等待输入一个数，并赋值给 x
    print('30 除以',x,'等于',30/x)         #输出 30 除以 x
except (ZeroDivisionError,ValueError):    #同时捕获 ZeroDivisionError 与 ValueError 异常
    print('输入错误，重新输入.....')
except :                                   #捕获其他异常
    print('其他异常.....')
```

上面代码中，将两个异常（ZeroDivisionError 与 ValueError）放到了一个处理分支下。程序运行后，当输入 a（非数字）或 0 时，都会输出“输入错误，重新输入.....”。

7.2.2 异常处理中的 else 语句

try……except……语句后面还可以跟 else 语句。当没有异常发生时，将执行 else 语句。else 语句是个可选语句，必须要放在所有 except 语句后面。例如：

```
try:
    x = int(input('请输入一个被除数:'))    #等待输入一个数，并赋值给 x
    print('30 除以',x,'等于',30/x)         #输出 30 除以 x
except (ZeroDivisionError,ValueError):    #同时捕获 ZeroDivisionError 与 ValueError 异常
    print('输入错误，重新输入.....')
except :                                   #捕获其他异常
    print('其他异常.....')
else:
    print("再见")
```

执行上面程序，当输入整数 6 时显示如下内容：

```
请输入一个被除数:6
30 除以 6 等于 5.0
再见
```

程序走完了正常流程，没有产生异常。执行了 else 中的内容，于是输出“再见”。

7.2.3 输出未知异常

前面的 7.1.1 小节中的例子，直接捕获了指定的异常类型。这是因为程序比较简单，可以预知会发生哪些异常。如果在一些复杂的程序中，则往往很难预测有什么异常情况发生。这时可以使用如下语法输出未知异常。

```
try:
```

```
.....some functions.....
except Exception as e:
    print(e)
```

通过 `except` 后面跟着 `Exception as e` 的语句，可得到异常类型 `e`。下面通过代码演示：

```
try:
    x = int(input('请输入一个被除数:'))          #等待输入一个数并赋值给 x
    print('30 除以',x,'等于',30/x)              #输出 30 除以该输入的数字
except Exception as e:                          #捕获未知异常
    print(e )
    print('其他异常.....')
```

执行上面程序，当输入整数 0 时显示如下内容：

```
division by zero
其他异常.....
```

可以看到，程序打印出了异常类型的原因——除数为 0。

7.2.4 输出异常的详细信息

在实际调试程序的工作中，只获得异常的类型是远远不够的。有时还必须借助更详细的异常信息，辅助开发人员才能解决问题。

捕获异常时，可以使用 `sys` 模块中的 `exc_info` 函数，或使用 `traceback` 模块中的相关函数，来获得更多的异常信息。详细介绍如下：

1. `sys` 的 `exc_info` 函数

模块 `sys` 中有两个函数可以返回异常的全部信息：一个是 `exc_info`；另一个是 `last_traceback`。两个函数有相同的用法及功能。这里以 `exc_info` 为例。

`exc_info` 函数会将当前的异常信息以元组的类型返回。元组内包含 3 个元素，分别为 `type`、`value` 和 `traceback`。

- **Type:** 异常类型的名称，它是 `BaseException` 的子类（类与子类见第 9 章的详细介绍）。
- **value:** 捕获到的异常实例（“实例”的概念请见第 9 章的详细介绍）。
- **traceback:** 是一个 `traceback` 对象（见本节的“2. `traceback` 对象的显示”）。

下面用代码演示函数 `exc_info` 的使用方法。

```
import sys
```

```

try:
    x = int(input('请输入一个被除数:'))          #等待输入一个数
    print('30 除以',x,'等于',30/x)              #输出 30 除以该输入的数字
except :
    print(sys.exc_info() )                      #捕获其他异常
    print('其他异常.....')

```

运行程序上面的程序。输入 0，令程序发生异常。输出如下：

```

请输入一个被除数:0
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',), <traceback
object at 0x000000000C745748>)
其他异常.....

```

输出结果的第 2 行为异常的全部信息。该信息是一个元组类型。元组的第 1 个元素是一个 `ZeroDivisionError` 类；第 2 个元素是异常类型 `ZeroDivisionError` 类的一个实例；第 3 个元素为一个 `traceback` 对象。

通过这种输出，不仅可以看到异常类型的信息，还可以从 `traceback` 对象里面找到异常位置的调用堆栈信息，具体请参见下面的“2. `traceback` 对象的显示”对 `traceback` 模块的介绍。

2. `traceback` 对象的显示

要查看 `traceback` 对象的内容，需要先引入 `traceback` 模块，调用 `traceback` 模块中的 `print_tb` 方法，并将 `sys.exc_info` 输出的 `traceback` 对象传入。`print_tb` 方法专用于显示 `traceback` 对象的内容。例如：

```

import traceback
import sys
try:
    x = int(input('请输入一个被除数:'))          #等待输入一个数
    print('30 除以',x,'等于',30/x)              #输出 30 除以该输入的数字
except :
    traceback.print_tb(sys.exc_info()[2])        #打印 traceback 对象
    print('其他异常.....')
else:
    print("再见")

```

运行程序上面的程序。输入 0，令程序发生异常。输出如下：

```

其他异常.....
File "<i>Python-input-6-74537bd39c61</i>", line 5, in <module>
    print('30 除以',x,'等于',30/x)              #输出 30 除以该输入的数字

```

这次程序输出了更多的异常信息，包括文件名、行数、模块名、代码（见输出结果的第二行），这些都是发生异常的现场环境信息。

3. traceback 模块的其他函数

通过 traceback 模块的载入，可以更简单地获得更多异常信息显示。比如调用 traceback 对象的 print_exc 方法，可以直接将异常内容打印出来。例如：

```
import traceback
try:
    x = int(input('请输入一个被除数:'))          #等待输入一个数
    print('30 除以',x,'等于',30/x)              #输出 30 除以该输入的数字
except :                                          #捕获其他异常
    traceback.print_exc()
    print('其他异常.....')
else:
    print("再见")
```

运行程序上面的程序。输入 0，令程序发生异常。输出如下：

```
请输入一个被除数:0
其他异常.....
Traceback (most recent call last):
  File "<iPython-input-4-a9ad416eade4>", line 4, in <module>
    print('30 除以',x,'等于',30/x)
ZeroDivisionError: division by zero
```

可以看到打印的信息中包含了程序出错的位置（见代码的第 4、5 行）、异常类型（见代码的最后一行）、异常类型的说明（见代码的最后一行）。

在 traceback 模块中，还有功能类似的 print_exception 函数。但是 print_exception 需要传入 sys.exc_info() 的结果。即，下面两行代码是等价的：

```
traceback.print_exc()
traceback.print_exception(*sys.exc_info())
```

7.3 捕获与处理异常

异常处理的过程中，try 与 except 语句都做了哪些操作？这得从异常的处理流程与 try 的工作原理说起。

7.3.1 异常的处理流程

try 语句在执行时，如果出现了一个异常，该语句的剩余部分将被跳过；如果该异常的类型匹配到了 except 后面的异常名，则该 except 后的语句将被执行。



注意：

如果 except 后面没有跟异常名，但必须放在所有捕获异常的语句之后，表示它可以匹配任何类型的异常。

7.3.2 try 语句的工作原理

try 语句的工作原理如下：

- (1) 当执行一个 try 语句后，Python 就在当前程序的上下文中做标记。
- (2) 如果程序出现异常，系统就会回到标记处。接着执行第一个匹配该异常的 except 子句。异常处理完毕，控制流就跳过整个 try 语句（除非在处理异常时又引发新的异常）。
- (3) 如果程序出现异常，但没有找到相匹配的 except 子句。异常将被递交到上层的 try 或者程序的最上层（这样将结束程序，并打印默认的出错信息）。
- (4) 如果 try 子句在执行时没有发生异常，Python 将执行 else 后的语句（如果有 else 的话），然后控制流跳过整个 try 语句。

7.3.3 一些常见的异常

在 Python 中，内置了一些常见的异常，即标准异常。标准异常的具体类型见表 7-1。

表 7-1 标准异常

| 异常名称 | 描 述 |
|-------------------|-------------------------|
| BaseException | 所有异常的基类 |
| SystemExit | 解释器请求退出 |
| KeyboardInterrupt | 用户中断执行（通常是输入^C） |
| Exception | 常规错误的基类 |
| StopIteration | 迭代器没有更多的值 |
| GeneratorExit | 生成器（generator）发生异常来通知退出 |
| StandardError | 所有的内建标准异常的基类 |

续表

| 异常名称 | 描 述 |
|-----------------------|-----------------------------------|
| ArithmeticError | 所有数值计算错误的基类 |
| FloatingPointError | 浮点计算错误 |
| OverflowError | 数值运算超出最大限制 |
| ZeroDivisionError | 除（或取模）零（所有数据类型） |
| AssertionError | 断言语句失败 |
| AttributeError | 对象没有这个属性 |
| EOFError | 没有内建输入，到达 EOF 标记 |
| EnvironmentError | 操作系统错误的基类 |
| IOError | 输入/输出操作失败 |
| OSError | 操作系统错误 |
| WindowsError | 系统调用失败 |
| ImportError | 导入模块/对象失败 |
| LookupError | 无效数据查询的基类 |
| IndexError | 序列中没有此索引（index） |
| KeyError | 映射中没有这个键 |
| MemoryError | 内存溢出错误（对于 Python 解释器不是致命的） |
| NameError | 未声明/初始化对象（没有属性） |
| UnboundLocalError | 访问未初始化的本地变量 |
| ReferenceError | 弱引用（Weak reference）试图访问已经垃圾回收了的对象 |
| RuntimeError | 一般的运行时错误 |
| NotImplementedError | 尚未实现的方法 |
| SyntaxError | Python 语法错误 |
| IndentationError | 缩进错误 |
| TabError | Tab 和空格混用 |
| SystemError | 一般的解释器系统错误 |
| TypeError | 对类型无效的操作 |
| ValueError | 传入无效的参数 |
| UnicodeError | Unicode 相关的错误 |
| UnicodeDecodeError | Unicode 解码时的错误 |
| UnicodeEncodeError | Unicode 编码时错误 |
| UnicodeTranslateError | Unicode 转换时错误 |
| Warning | 警告的基类 |
| DeprecationWarning | 关于被弃用的特征的警告 |
| FutureWarning | 关于构造将来语义会有改变的警告 |


```

        raise ValueError('输入错误: 0 不能做被除数')
    print('30 除以',x,'等于',30/x)           #输出 30 除以该输入的数字
except Exception as e:
    print(e )
except ZeroDivisionError:                    #捕获 ZeroDivisionError 异常
    print('被除数不等于 0, 重新输入……')
except :                                     #捕获其他异常
    print('其他异常……')

```

这段代码是让用户输入一个被除数，并用 30 除以这个输入的被除数。由于 0 不能作为除数，于是在得到输入数值之后加了一个判断。如发现输入为 0，则由程序主动生成一个异常。在后面代码中，通过 `except` 关键字来捕获该异常。

将程序运行起来，输入 0，则会有如下输出：

```

请输入一个被除数:0
输入错误: 0 不能做被除数

```

可以看到，在捕获异常时，程序运行了 `ValueError` 异常分支。这与代码中通过 `raise` 生成的异常类型（`ValueError`）一致。在 `ValueError` 异常分支中，执行的代码是将异常信息打印出来，于是程序就输出“输入错误：0 不能做被除数”。

使用 `raise` 语句的程序会更加严谨，可以在程序执行到错误的分支情况时主动报出异常，增强了程序逻辑的健壮性。在出错时，还可以提供自定义的人机交互信息，使程序调试起来，更为方便。

7.5 异常的最终处理（清理动作）

Python 中，`finally` 语句是与 `try` 和 `except` 语句配合使用的。

`finally` 语句中的内容一般都是用来做清理工作的。无论 `try` 中的语句是否跳入 `except` 中，最终都要进入 `finally` 语句，并执行 `finally` 语句的分支代码。

7.5.1 `finally` 的使用场景

`try` 和 `except` 语句改变了程序执行的流程。代码在没有运行的前提下，并不知道会进入哪个分支。在复杂的逻辑关系中，很容易出现由于出现资源不释放，而引起资源泄漏或内存泄漏的问题。

为了解决这种问题，一般会在 `except` 的最后加个 `finally`，并在 `finally` 语句中将资源或是内存进行统一地清理。这样，无论程序运行到哪个分支，最终都会进入 `finally` 语句的分支代码进行资源的收尾工作，从而避免了资源或内存泄漏的问题。例如：

```
try:
    print('打开一个文件')           #伪码：打开一个文件
    print('读取内容')               #伪码：假设打开成功，开始读取文件
    raise IOError('读取出错')      #伪码：假设在读取过程中发生了错误
except Exception as e:              #捕获错误异常
    print(e)
finally:                             #无论程序是否错误，执行完前面代码后都要在这里关闭文件
    print("关闭文件")
```

例子中，通过伪码来描述一个打开并读取文件的过程（文件的真实操作在第 8 章会有介绍）。在真正执行时，会发生读取正常和读取出错两种情况。无论哪种情况，都会进入 `finally` 分支。在 `finally` 分支中，将已经打开的文件关闭，以确保资源得到释放。

7.5.2 finally 与 else 的区别

`finally` 与 `else` 的区别在于：

- `else` 语句只在没有异常发生的情况下执行。
- `finally` 语句则不管异常发生与否都会执行。

`finally` 语句总是在退出 `try` 语句前被执行，无论是正常退出、异常退出，还是通过 `break`、`continue`、`return` 语句退出。

7.6 判定条件的正确性（断言）

Python 中的 `try` 与 `except` 语句一般是用来捕捉用户或者环境的错误。而断言，是人为的判定条件的正确性，即，检验自己的判断是对还是错。

7.6.1 断言的表达形式

断言，使用 `assert` 关键字后面接着一个条件表达式。

- 如果条件表达式为真，意味着程序当前的条件与开发人员自己断言的情况一致，则程序继续运行。

- 如果为假，则表明一定是前面发生了错误，程序停止运行，报出异常。

例如：

```
assert 1!=1          #断言 1 不等于 1
```

这句代码断言“1 不等于 1”，这显然是个错误的条件。于是报错：

```
File "<iPython-input-4-c5230ec50e34>", line 1, in <module>
    assert 1!=1
AssertionError
```

7.6.2 带错误信息的断言语句

Python 中，如果断言后面的条件语句失败了，还可以为其指定对应的字符串输出。这样代码就会变得更加友好。具体做法是：直接在 `assert` 后面的条件语句之后，加上“`, 字符串`”。

例如：

```
assert 1!=1, ("1 不等于 1, 报错")      #断言 1 不等于 1
```

执行这段代码时，输出的结果就会更加人性化一些：

```
File "<iPython-input-4-c5230ec50e34>", line 1, in <module>
    assert 1!=1
AssertionError: 1 不等于 1, 报错
```

从输出结果的最后一行可以看到，`AssertionError` 后面会显示出来详细的出错信息。

在正规的开发流程中，单元测试是每个程序员都应该做的事情。开发人员编写完代码后，可以利用断言语句来检查自身的错误，实现单元测试。断言常常被用在单元测试和参数检查中，以避免程序运行时出现开发人员编写中的逻辑错误。

7.7 实例 20：如 HTTP 请求失败，实现“重试”功能

在自动化爬虫项目中，通常会对几十甚至上百个 URL 进行批量爬取。在对某一个具体的 URL 爬取的过程中，第一步就是发送 HTTP 请求。在请求时，可能由于网络不稳定，有时即使是正确的 URL 也会返回请求失败。这时需要对该 URL 重新发送请求，以确保不漏爬该 URL 的内容。经过几次重试之后，如果仍然返回请求失败，则认定该 URL 失效，开始爬取下一个 URL。

下面就来通过实例演示 HTTP 请求失败后的“重试”功能。

实例描述

编写代码，按照如下情况实现 HTTP 请求，并将其返回内容打印出来。

- (1) 爬取“163”的网页，并将返回内容打印出来。
 - (2) 爬取一个错误的 URL 网页，当请求失败时，再次发出请求。重复 3 次，直到成功为止。
 - (3) 爬取“163”的网页，在发送请求的同时断网。当第 1 次请求失败时连上网，观察效果。
-

本实例实现 3 种情况的实验：

- 第 1 种情况是，正常的爬取行为。
- 第 2 种情况是，URL 失效的异常。
- 第 3 种情况是，由于网络原因引起的请求失败。

由于第 2 种与第 3 种情况触发的处理机制一样，这里只演示第 2 种。有兴趣的读者可以自行演示第 3 种。

7.7.1 使用装饰器实现失败重试

这部分知识属于第 6 章函数内容。主要思路如下：

- (1) 定义一个装饰器函数 `retry`，将重试次数作为函数参数。
- (2) 在 `retry` 中，使用 `while` 循环来进行请求处理，并记录请求次数。
- (3) 请求处理过程中，使用 `try` 语句来运行被装饰的请求网络函数，当 HTTP 请求失败时，通过 `except` 捕获异常，并调整请求次数。

代码如下：

代码 7-1：实现 HTTP 请求失败过程中的重试功能

```
01 def retry(attempt):                                #定义装饰器函数
02     def decorator(func):
03         def wrapper(*args, **kw):
04             att = 0
05             while att < attempt:                    #按照计数器att的条件来执行循环语句
06                 print(att)
07                 try:                                #使用 try except 捕获异常
08                     return func(*args, **kw)        #运行请求
09                 except Exception as e:
10                     att += 1                          #调整计数器
11             return wrapper
12     return decorator
```

这部分内容在第 6 章已经有详细的介绍。如果读者不太熟，可以自行复习第 6 章内容。

7.7.2 编写简单爬虫

编写简单爬虫的主要思路如下：

(1) 定义一个函数 `get_response`，并实现爬虫功能。

(2) 导入 HTTP 的网络请求模块 `requests`，并调用 `requests` 模块中的 `get` 函数，实现 HTTP 请求。

具体代码如下：

代码 7-1：实现 HTTP 请求失败过程中的重试功能（续）

```
13 import requests                                #导入网络请求模块
14
15 @retry(attempt=3)                               #应用装饰器并定义重试次数
16 def get_response(url):                          #定义爬虫函数
17     r = requests.get(url)                       #对指定 URL 发出 HTTP 请求
18     return r                                    #返回请求结果
```

函数 `get_response` 在装饰器 `retry` 的作用下，即可实现“重试”功能。

7.7.3 传入正确的目的地址，开始爬取

定义 URL 变量为 163 网站的网址，调用函数 `get_response` 发送 HTTP 请求，并将返回结果打印。

代码 7-1：实现 HTTP 请求失败过程中的重试功能

```
19 URL = 'http://www.163.com'
20 r = get_response(URL2)
21 print(r)
```

上面代码运行后输出如下内容：

```
<Response [200]>
```

200 是返回值，表明该网页返回正确的结果。添加如下代码，将返回的网页打印出来。

代码 7-1：实现 HTTP 请求失败过程中的重试功能（续）

```
22 print(r.content.decode('gbk'))
```

该代码的意思是读取返回值的 `content` 内容，并将其转为 GBK 编码，然后输出到屏幕。运行代码，输出如下结果：

```
.....
    [{i:"KK1",f:1}]]}}],".hot_pop":[{"":[{"":[{i:"L11"}]]}],".N-nav-bottom":[{".nte
s_foot_link":[{"":[{i:"M11",f:1}]]}],".feedback_close":[{"":[{i:"M21"}]]}],".#fixed_
foot_ad":[{"":[{"":[{i:"N11"}]]}]]}],le3)){});</script>
    <!-- 性能监测 运维-->
    <script      src="https://static.ws.126.net/common/script/meter.min.js"      type=
"text/javascript"></script>
    <script type="text/javascript">
        if(Math.random() < 0.1) {
            (new Meter({project: '网易首页',version: '1.0'})).start();
        }
    </script>
</body>
</html>
```

由于网页太长，这里只节选了尾部内容显示。

7.7.4 传入错误的目的地址，验证“重试”功能

定义变量 `URL2` 为一个错误的地址，并通过 `get_response` 函数向其发送请求。编写代码如下：

代码 7-1：实现 HTTP 请求失败过程中的重试功能（续）

```
23 URL2 = 'http://www.1632334434.com'
24 r = get_response(URL2)
25 print(r)
```

上面代码运行后，输出如下内容：

```
0
1
2
None
```

输出结果的前 3 行为数字 0、1、2，它们代表重试请求的次数，对应于代码的第 6 行 `print(att)`。输出结果的最后一行为 `None`，是函数 `get_response` 的返回值。因为程序请求的是个错误的地址，所以得到了一个结果为 `None` 的返回值。程序在 HTTP 请求失败后，又重试了 3 次，均失败，最后自动返回。

第 8 章

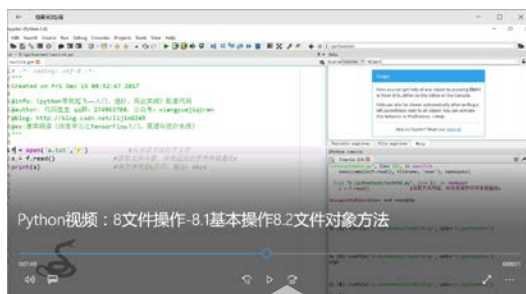
文件操作——数据持久化的一种方法

数据持久化，就是将 Python 程序中的对象以数据的方式存储在磁盘上，便于以后读取。在程序下次运行时，可以将数据从磁盘上恢复到内存中来。

有多种方法可以实现数据持久化功能，其中最常见的方法是将数据以文件的形式保存。在 Python 中，可以通过调用内置函数的方法进行文件的建立、读、写、删除等操作。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解

具体代码操作演示



python视频：8
文件操作-8.1基
本操作8.2文件对
象方法.mp4



python视频：8
文件操作-8.3实
例8.4with语句.
mp4



python视频：8
文件操作-8.5二
进制转化8.6序
列化.mp4

本章共有 3 段教学视频，总时长为 30 min 左右。包含下内容：

- 讲述了文件的基本操作、文件对象的方法。
- 演示了一个带有异常处理文件操作的例子，讲述了使用 `with` 语句简化代码的方法及原理，并演示了一个使用 `with` 语句操作文件的例子。
- 讲述了字符串与二进制的相互转化的方法，以及将任意对象序列化的方法。

8.1 文件的基本操作

文件的操作有很多种，例如：创建、删除、修改权限、写入、读取等。

- 删除、修改权限：作用于文件本身，属于系统级操作。
- 写入、读取：是文件最常用的操作，作用于文件的内容，属于应用级操作。

文件的系统级操作功能单一，容易实现。编码时，可以导入 Python 中的专用模块（`os`、`sys` 等），并调用模块中的指定函数来实行。例如，假设代码的同级目录下有一个文件“`a.txt`”，可以直接调用 `os` 模块下的 `remove` 函数将其删除，具体代码如下：

```
import os
os.remove('a.txt')          #删除文件 a.txt
```

执行代码后，本地的 `a.txt` 文件将会被删掉。

对于文件的应用级操作，是有固定步骤的，实现起来相对比较复杂。下面重点讲解文件应用级操作的细节。

8.1.1 读写文件的一般步骤

读写文件可以分为 3 步，每一步有对应的函数。

- （1）打开文件：使用 `open` 函数，返回的是一个文件对象。
- （2）具体读写：使用该文件对象的 `read`、`write` 等方法。
- （3）关闭文件：使用该文件对象的 `close` 方法。

一个文件，必须在打开之后才可以对其进行操作，并在操作结束之后将其关闭。这三步的顺序不能打乱。下面就来介绍相关函数的使用方法及细节。

8.1.2 打开文件

“打开”是文件读写操作的第一步。`open` 函数的具体定义如下：

```
open ( 文件名, mode )
```

函数中有两个参数。

- 文件名：属于字符串类型。使用时要注意转义问题，尽可能使用源字符串（以 `r` 开头的字符串，参见 4.3 节）。
- Mode：是指打开文件的方式，包括只读、只写、读写、二进制等。如果不指定 `mode` 参数，文件将默认以“只读”模式打开。

函数 `open` 的返回值是一个文件对象。该对象中封装了文件的各种操作。

1. `open` 中的模式介绍

在 `open` 函数中，参数 `mode` 起主要作用。它决定了文件的打开模式。具体如下：

- `r`：只读。文件必须存在。
- `w`：只写。如果文件已存在，则将其覆盖。如果该文件不存在，则创建新文件。
- `+`：读写（不能单独使用）。
- `a`：以只写的方式打开文件，用于在文件后追加内容。如果文件不存在，则创建新文件。
- `b`：以二进制模式打开（不能单独使用）。

在上面列出的 `mode` 值中，只有 `w` 和 `a` 可以创建文件。在实际应用中，这些 `mode` 值还可以组合使用，即同时使用多种模式来操作文件。调用 `open` 函数时，传入 `mode` 的常用值有 `r`、`w`、`r+`、`w+`、`rb`、`wb`、`rb+`、`wb+`、`a`、`a+`、`ab`、`ab+`。



注意：

`r+`、`w+`、`a+`都是可读写的意思。三者的区别是：

- `r+`：读写。文件必须存在。当写入时，会清空原内容。
- `w+`：读写。如果该文件不存在，则创建新文件。如果文件已存在，则清空原有内容。
- `a+`：读写。如果文件不存在，则创建新文件。如果文件已存在，则在文件后面追加内容。

通常情况下，文件都是以文本模式（`text mode`）打开的。即，从文件中读写的是以一种特定的编码格式（默认的是 `UTF-8`）进行编码的字符串。如果文件以二进制模式（`binary mode`）打开，则数据将以字节对象的形式进行读写。

2. 文本模式与二进制模式的区别

文本模式与二进制模式的区别如下：

- 在 Windows 系统中，文本模式下，默认是将 Windows 平台的行末标识符 `\r\n` 在读取操作时转为 `\n`，而在写入操作时将 `\n` 转为 `\r\n`。这种隐藏的行为对于文本文件是没有问题的，但如果以文本模式打开二进制数据文件（如 JPEG 或 EXE）则会发生问题，因为它改变了具体内容。
- 在 Unix/Linux 系统中，行末标识符为 `\n`，即文件以 `\n` 代表换行。所以，在 Unix/Linux 系统中文本模式和二进制模式并无区别。

3. 函数 open 的返回对象

函数 open 的返回值是由打开模式决定的，具体如下：

- 文本模式：返回 `TextIOWrapper` 对象。
- 读取二进制模式：即 “`r+b`” 模式，返回 `BufferedReader` 对象。
- 写入和追加二进制模式：即 “`w+b`” “`a+b`” 模式，返回 `BufferedWriter` 对象。
- 读/写模式：即含有符号 “`+`” 的打开模式，返回 `BufferedRandom` 对象。

8.1.3 具体读写

通过 open 函数得到文件对象后，就可以对文件进行操作了。最常用的方式是读和写。下面分别举例。

1. 读取文件

通过调用文件对象的 `read` 方法可以获得文件的内容。

比如，在代码的同级目录下，新建一个文本文件，名为 “`a.txt`”，并向 “`a.txt`” 中写入字符 “`abcd`”。可以通过如下代码读出其中的内容。

```
f = open('a.txt', 'r')           #以只读方式打开文件
s = f.read()                     #读取文件内容，并将返回的字符串赋值给 s
print(s)                         #将文件内容 s 打印，输出：abcd
```

文件对象 `f` 的 `read` 方法，会将文件的全部内容一次性读取到内存中。

2. 写入文件

将字符串写入文件，可以调用文件对象的 `write` 方法。

比如，在代码的同级目录下，新建一个文本文件“a.txt”。可以通过如下代码向文件中写入字符“efgh”。

```
f = open('a.txt','w')      #以写入的方式打开文件
f.write('efgh')             #写入 efgh
f = open('a.txt','r')      #以只读的方式打开文件
s = f.read()                #读取文件内容，并将返回的字符串赋值给 s
print(s)                    #将文件内容 s 打印，输出：efgh
```

代码的前两行，完成了一个打开文件并写入内容的操作。后三行则将文件内容读入内存，并显示出来。

屏幕输出了 `efgh`，与写入的内容完全相同。



注意：

如果文件是以二进制形式打开的，则只能以二进制形式写入，否则会报错。例如：

```
f = open('a.txt','wb+')      #以二进制形式打开一个文件
f.write('I like Python!')    #以文本形式向该文件写入数据，会报错
```

上面的代码中，先以二进制形式打开一个文件，接着以文本的方式向该文件中写入一个字符串。这样会报错。写入时，必须将字符串转成二进制数。正确的写法如下：

```
f = open('a.txt','wb+')
f.write(b'I like Python!')    #正确的写法是，以 bytes 对象的形式进行读写
```

上面代码中，在字符串前面加一个 `b`，代表该字符串是二进制形式。这时再通过 `write` 进行写入，则成功。

关于文件的读写远不止 `read` 与 `write` 函数，更多具体操作参见 8.2 节。

8.1.4 关闭文件

直接使用文件对象的 `close` 方法可关闭文件。文件在打开并操作完事之后，需要及时关闭，否则会給程序带来好多无法预知的错误。

例如，对一个没有关闭的文件进行删除操作，会失败。代码如下：

```
import os
f = open('a.txt', 'wb+')          #以二进制形式打开一个文件
f.write(b'I like Python!')        #以二进制形式进行写入
os.remove('a.txt')                #要删除文件 a.txt，但是该文件还没有被关闭
```

文件还没有进行关闭，是无法被删除的。上面代码执行后，会提示如下错误：

```
PermissionError: [WinError 32] 另一个程序正在使用此文件，进程无法访问。: 'a.txt'
```

同时，可以看到当前代码的同级目录的下，生成了一个 a.txt 文件。但是打开该文件会发现其中并没有内容。这表明文件对象的 write 方法只是把当前的内容缓存到了内存里，并没有真正写入到文件。

如果加上一句 close 再执行，就会看到文件 a.txt 中有了内容。具体代码如下：

```
f = open('a.txt', 'wb+')          #以二进制形式打开一个文件
f.write(b'I like Python!')        #以二进制形式进行写入
f.close()                        #关闭文件
```

上面代码执行后，再打开当前代码的同级目录的下 a.txt 文件，发现里面已经有内容了。这表明在调用 close 时，系统自动将缓存里的内容写进了文件。



注意：

如果读者使用的是 Spyder 开发环境，在出现 PermissionError 异常后，会发现如下现象：即使修改代码，添加“关闭”功能，也同样会提示 PermissionError 异常错误。这时需将当前的控制台（consoles）关闭，再重新打开一个 consoles，这样才可以正常运行。

8.2 文件对象的方法

前面 8.1.3 小节中提到文件对象的操作不仅仅是 read 与 write 两个方法，还有更多的方法，可以使用 help 或 dir 函数对文件对象进行查询。

8.2.1 文件对象的常用方法介绍

表 8-1 列出了一些常用的方法（假设 f 是一个文件对象）。

表 8-1 标准异常

| 文件操作方法名称 | 描 述 |
|---------------------------|--|
| f.read(size) | 读取 size 个字节的数据，然后作为字符串或 bytes 对象返回。size 是一个可选参数。如果不指定 size，则读取文件的所有内容 |
| f.readline() | 读取一行，并在字符串末尾留下换行符 (\n)。如果到达文件尾，则返回空字符串 |
| f.readlines() | 读取所有行，并储存在列表中。每个元素是一行，相当于 list(f) |
| f.write(string) | 将 string 写入到文件中，返回写入的字符数。如果以二进制模式写文件，则需要将 string 转换为 bytes 对象 |
| f.tell() | 返回文件对象当前所处的位置。是从文件开头开始算起字节数 |
| f.seek(offset, from_what) | 改变文件对象所处的位置。offset 是相对参考位置的偏移量。from_what 表示参考位置，取值为 0（文件头，默认）、1（当前位置）、2（文件尾） |

在文本模式下，定位文件尾可直接使用 seek(0, 2)语句；定位文件头可直接使用 seek（0）语句。

8.2.2 把文件对象当作迭代器来读取

文件对象本身也是一个迭代器，它也可以与 for 循环配合进行文件的读取，例如：

```
f = open('a.txt', 'wb+') #以二进制形式打开一个文件
f.write(b'I like Python!\n') #以二进制形式进行读写
f.write(b'I love Python!') #再写一行
f.close() #关闭文件
f = open('a.txt', 'r+') #打开文件
for line in f: #直接使用 for 循环读取文件
    print(line) #将内容打印出来
f.close() #关闭文件
```

上面的代码，先往文件中写入两行字符串，然后使用 for 循环来遍历 open 返回的文件对象。注意，这里没有使用文件对象的任何内置方法，同样也可以将文件内容读取出来。程序执行后，输出如下：

```
I like Python!

I love Python!
```

这种使用 for 循环的写法，相当于在每一次迭代中都调用了 一个 readline 方法。

8.3 实例 21：带有异常处理的文件操作

在 7.5.1 小节里，举了一个打开文件的例子，当时用的是伪代码。这里使用真实代码来实现一个文件操作配合异常处理的案例。

实例描述

对文件分别进行写入和读取操作，并且都使用 `try/except` 进行异常处理，最终在 `finally` 中进行关闭。

(1) 在第一个 `try` 里面，以二进制形式打开一个文件，以文本的方式向里面写入一个字符串。使用 `except` 对 `try` 里面的异常进行捕获。最终在 `finally` 中对文件进行关闭。

(2) 在第二个 `try` 里面，以文本的方式打开一个文件，并读取里面的内容。使用 `except` 对 `try` 里面的异常进行捕获。最终在 `finally` 中对文件进行关闭。

第一个 `try` 里故意放置了一段错误代码，以文本的方式写入一个用二进制打开的文件是会报错的，程序会跳到 `except` 的异常捕获分支里，最终在 `finally` 中关闭文件。

第二个 `try` 则是正确的代码，程序不会报异常，但也会进入 `finally` 模块关闭文件。

代码 8-1：文件操作配合异常处理

| | |
|---|------------------------------|
| <code>try:</code> | #第一个try语句 |
| <code>f = open('a.txt','wb+')</code> | #以二进制的形式打开一个文件 |
| <code>f.write('I like Python!')</code> | #以文本的方式写入一个用二进制打开的文件，会报错 |
| <code>except Exception as e:</code> | #将错误异常捕获 |
| <code>print(e)</code> | |
| <code>f.write(b'I like Python!')</code> | #以二进制的形式进行读写 |
| <code>finally:</code> | #无论程序是否错误，在执行完前面代码后都要在这里关闭文件 |
| <code>print("关闭文件")</code> | |
| <code>f.close()</code> | #关闭文件 |
| | |
| <code>try:</code> | #第二个try语句 |
| <code>f = open('a.txt','r+')</code> | #打开文件 |
| for line in f: | #直接使用for循环读取文件 |
| <code>print(line)</code> | #将内容打印出来 |
| <code>finally:</code> | |
| <code>f.close()</code> | #关闭文件 |

由于第一个 `try` 中的代码在写入文件时发生了异常，导致文件没有成功写入，于是在 `except` 中，以正确的方式将内容写入文件中。代码执行后显示如下：


```
a bytes-like object is required, not 'str'  
关闭文件  
I like Python!
```

显示的结果解释如下：

- 第 1 行，所报的异常。
- 第 2 行，进入 `finally` 关闭文件时的打印内容。
- 第 3 行，将文件中的内容读取出来并打印。

这个例子演示了打开文件的正确方式。在实际代码开发中，尤其像文件操作，这种对环境依赖很强的操作，非常建议使用异常处理的方式来操作。这也是编写健壮程序的好习惯。

8.4 使用 with 语句简化代码

在 Python 编程中，有很多代码流程都与文件操作的步骤类似。如果将操作文件看作一个任务，该任务具有事先、事中、事后明显的三个阶段。即，事先需打开文件（`open`），事中需进行文件操作，事后需关闭文件（`close`）。

Python 中内置了 `with` 语句，可以使代码更加简化。`with` 语句适用于类似文件操作的这种（具有事先、事中、事后三个明显阶段）任务。

使用 `with` 语法时，只需关心事先、事中的事情，可以不关心事后事情。`with` 语句可以让文件对象在使用后被正常关闭。

8.4.1 实例 22：使用 with 语句操作文件

将 8.3 节的实例改用 `with` 语句实现。

实例描述

使用 `with` 语句对文件分别进行写入和读取操作，并且使用 `try/except` 进行异常处理。

（1）在第一个 `with` 里，以二进制形式打开一个文件，以文本的方式向其中写入一个字符串，并使用 `try/except` 语句捕获异常。

（2）在第二个 `with` 里，以文本的方式打开一个文件，并读取里面的内容，并使用 `try/except` 语句捕获异常。

`with` 语句的写法如下：

with 表达式 as 变量
具体的操作语句

其中，表达式就是 open 函数，as 后面的变量就是 open 返回的文件类型。



注意：

with 语句也是有作用域的，作用域的代码同样通过缩进的方式来表示。当 with 作用域内的语句执行完毕后，就会自动调用 f 的 close 方法将文件关闭。代码如下：

代码 8-2：演示 with 语法操作文件

```
with open('a.txt', 'wb+') as f:      #以二进制形式打开一个文件
    try:
        f.write('I like Python!')    #以文本的方式写入一个用二进制打开的文件，会报错
    except Exception as e:           #将错误异常捕获
        print(e)
        f.write(b'I like Python!')  #以 bytes 对象的形式进行读写

with open('a.txt', 'r+') as f:      #打开文件
    for line in f:                  #直接使用 for 循环读取文件
        print(line)                #将内容打印出来
```

可以看到，使用 with 语句可以实现 8.3 节中例子同样的效果，但是代码少了很多。最主要的是，它把事后的清理工作做成了自动完成，这在某种程度上降低了开发人员疏忽导致程序出错的可能。

8.4.2 with 语法的原理

在 Python 中，支持 with 语法的对象必须有一个 __enter__ 方法和一个 __exit__ 方法。

在 with 语法执行过程中，紧跟 with 后面的语句被求值后，返回对象的 __enter__ 方法被调用，这个方法的返回值将被赋值给 as 后面的变量。当 with 后面的代码块全部被执行完后，将调用前面返回对象的 __exit__ 方法。

之所以 with 对文件对象有效，是因为文件对象在其 __exit__ 方法里实现了“关闭”功能。

8.5 实现字符串与二进制数的相互转化

在处理文件操作时，将文件“以二进制的形式保存，以文本的方式使用”是一种常用的做法。因为文本被转化成二进制数后可以占用更小的空间，便于网络传输与硬盘存储。

二进制与文本相互转化会依赖于编码格式、操作系统等多个因素。如果处理不当，则无法得到想要的效果。

在 8.1.3 小节中，使用了一个例子“用二进制的形式打开一个文件，然后以文本字符串的方式往里写入，发生了错误”。当时的解决方法是，通过在一个字符串前面加上“b”来创建一个二进制对象。这里介绍创建二进制对象的另一种方法——使用 `bytes` 函数。

8.5.1 将字符串转二进制数

`bytes` 的定义如下：

```
bytes(字符串, 编码格式)
```

通过定义可以看出，使用 `bytes` 函数时必须同时提供一种编码格式。例如：

```
b1 = b'I like Python'          #在字符串前加 b，将其转为二进制
b2 = bytes('I like Python', 'UTF-8')  #使用 bytes 将其转为二进制
print(b1, b2, sep='; ')        #将结果打印出来，输出: b'I like Python'; b'I like Python'
```

8.5.2 将二进制数转字符串

如果将二进制转成字符串，可以调用二进制对象的 `decode` 方法，并传入指定的解码格式。例如：

```
b = bytes('I like Python', 'UTF-8')
print(b, b.decode(), sep='; ') #将结果打印出来，输出: b'I like Python'; 'I like Python'
```

上面的代码中，`decode` 方法没有传入指定的解码格式，表明使用了默认解码格式。当然也可以写成如下：

```
print(b, b.decode('UTF-8'), sep='; ') #将结果打印出来，输出: b'I like Python'; 'I like Python'
```

Windows 平台与 linux 平台上要使用不同的解码格式。

- 在 linux 平台下，生成的文件默认是 UTF-8 格式，所以需指定解码格式为 UTF-8。
- 在 Windows 平台下，生成的文件默认是 GB2312、GBK 等格式，所以需将其指定为对应的解码格式才可以正常地显示字符串。

8.6 将任意对象序列化

在 Python 中有个序列化过程叫作 `pickle`。它能够实现任意对象与文本之间的相互转化，也可以将任意对象与二进制之间的相互转化。即，可以透明地实现 Python 对象的存储及恢复。

使用 Python 的 `pickle` 操作，可以将对象序列化成字符串、磁盘上的文件等类似于文件的任何对象；也可以将这些字符串、文件或任何类似于文件的对象 `unpickle` 成原来的对象。

8.6.1 pickle 函数

导入 `pickle` 模块后，就可以实现 `pickle` 功能了。在 `pickle` 模块中，共有 4 个函数可以使用。

- `dumps`: 将 Python 中的对象序列化成二进制对象，并返回。
- `loads`: 从给定的 `pickle` 数据中读取并返回对象。
- `dump`: 将 Python 中的对象序列化成二进制对象，并写入文件。
- `load`: 读取指定的序列化数据文件，并返回对象。

这 4 个函数可以分成两类：`dumps` 与 `loads`，实现基于内存的 Python 对象与二进制互转；`dump` 与 `load`，实现基于文件的 Python 对象与二进制互转。

1. 将 Python 对象转成二进制（`dumps` 函数）

`dumps` 函数的具体定义如下：

```
dumps(obj, protocol=None, *, fix_imports=True)
```

参数说明如下。

- `obj`: 要转换的 Python 对象。
- `protocol`: `pickle` 的转码协议，取值为 0、1、2、3、4。其中，0 为 ASCII 码表示；2 为旧版本的二进制协议；3 为新的二进制协议；4 为更新的二进制协议。未指定情况下，默认为 3。
- 其他参数：是为了兼容以前 Python 2 版本而保留的参数，可以不管。

2. 将二进制对象转成 Python 对象（`loads` 函数）

`loads` 函数的定义如下：

```
loads(data, *, fix_imports=True, encoding='ASCII', errors='strict')
```

参数说明如下：

- **data**：要转换的二进制对象。
- **其他参数**：是为了兼容以前 Python 2 版本而保留的参数，可以不管。

在将二进制对象反序列化成 Python 对象时，会自动识别转码协议，所以不需要将转码协议当作参数传入。当待转换的二进制对象的字节数超过 pickle 的 Python 对象时，多余的字节将被忽略。

3. 将 Python 对象转成二进制文件（dump 函数）

dump 函数的定义如下：

```
dump(obj, file, protocol=None, *, fix_imports=True)
```

参数说明如下。

- **obj**：要转换的 Python 对象。
- **file**：文件必须有 write 方法，并且支持写入二进制数据。
- **protocol**：pickle 的转码协议，取值为 0、1、2、3、4。其中，0 为 ASCII 码表示；2 为旧版本的二进制协议；3 为新的二进制协议；4 为更新的二进制协议。未指定情况下，默认为 3。
- **其他参数**：是为了兼容以前 Python 2 版本而保留的参数，可以不管。

4. 将二进制对象文件转成 Python 对象（load 函数）

load 函数的定义如下：

```
load(file, *, fix_imports=True, encoding='ASCII', errors='strict')
```

参数说明如下：

- **File**：对象必须有两个方法——read()和 readline。
- **其他参数**：是为了兼容以前 Python 2 版本而保留的参数，可以不管。

在将二进制 unpickle 转成 Python 对象时，会自动识别转码协议。所以无需将转码协议当作参数传入。当待转换的二进制对象的字节数超过 pickle 的 Python 对象时，多余的字节将被忽略。

8.6.2 实例 23：用 pickle 函数实现元组与“二进制对象”“二进制对象文件”之间的转换

下面将举例演示 pickle 函数的使用。

实例描述

定义一个含有多种类型元素的元组，分别对其进行如下操作：

（1）使用 pickle 函数将元组转成二进制对象，然后再从二进制对象还原回来，观察其值是否有变化。

（2）使用 pickle 函数将元组转成二进制对象文件，然后再从二进制对象文件中还原回来，观察其值是否有变化。

具体操作如下。

1. 引入 pickle 模块，定义一个含有多种类型元素的元组

使用 import 引入 pickle 模块，定义一个元组，里面包含字符串、列表以及 None 类型。代码如下：

代码 8-3：pickle 函数的使用

```
01 import pickle                                #导入 pickle 模块
02
03 tup1 = ('I love Python', [1, 2, 3], None)    #定义一个含复杂元素的元组
```

tup1 就是准备做转换的数组。为了测试 pickle 的功能，这里使用了嵌套字符串和列表的复杂结构。

2. 实现 Python 对象与二进制对象的互转

先使用 dumps 函数将 tup1 转成 p1，再用 loads 函数将 p1 转回到 Python 对象，并赋值给 t2。代码如下：

代码 8-3：pickle 函数的使用（续）

```
04 #测试 Python 对象与二进制对象的互转
05 p1 = pickle.dumps(tup1)                #使用 pickle 模块的 dumps 将 Python 对象转成二进制对象
06 t2 = pickle.loads(p1)                  #使用 pickle 模块的 loads 将二进制对象转成 Python 对象
07
08 print(t2)                              #将还原的 Python 对象输出
```

上面代码的最后一行要输出还原后的 Python 对象。运行代码，得到如下输出：

```
('I love Python', [1, 2, 3], None)
```

可以看到，输出的结果与之前的元组对象 `tup1`（见第 3 行）值是一样的。

3. 实现 Python 对象与二进制对象文件间的互转

Python 对象与二进制对象文件互转，与“2. 实现 Python 对象与二进制对象的互转”中类似。只不过是，需要先将文件打开，再将文件对象与 `tup1` 一起传入 `dump` 函数。这时会生成一个文件 `a.pkl`。再次打开生成的文件 `a.pkl`，将文件对象 `f` 传入 `load` 函数中，解析出 Python 对象，并赋值给 `t3`。代码如下：

代码 8-3: pickle 函数的使用（续）

```
09 #测试 Python 对象与二进制文件互转
10 path = "a.pkl"
11 with open(path, 'wb') as f:                #打开文件
12     pickle.dump(tup1, f, pickle.HIGHEST_PROTOCOL)    #用 dump 将 Python 对象转成二进制
    对象文件
13
14 with open(path, 'rb') as f:                #再次打开文件
15     t3 = pickle.load(f)                    #用 load 将二进制对象文件转成 Python 对象
16     print(t3)                             #将还原的 Python 对象输出
```

上面代码第 12 行，`dump` 的第 2 个参数使用了一个宏定义 `pickle.HIGHEST_PROTOCOL`。其实 `pickle.HIGHEST_PROTOCOL` 的值就是 4，与直接填写 4 效果一样。

Python 对象与二进制对象文件互转时，必须提前打开文件。这里使用了 `with` 语句来打开文件。

程序运行后，得到如下输出：

```
('I love Python', [1, 2, 3], None)
```

可以看到，从文件转化出来的 Python 对象 `t3` 与原始的 `tup1` 对象是一样的。同时还可以看到，在本地硬盘上有了一个 `a.pkl` 文件。

如果读者以后从事深度学习相关工作，很多场景中都需要把样本或是中间态的训练数据保存起来。在这种情况下，如使用 `pickle` 技术则会使开发效率提高很多。

8.6.3 序列化的扩展方法（ZODB 模块）

看似强大的 pickle 模块，其实也有它的短板：它不支持并发地访问持久性对象（关于并发的知识在第 10 章会有详细地讲解）。在复杂的系统环境下，尤其是海量数据读取时，使用 pickle 会使整个系统的 I/O 读取性能成为瓶颈，必须使用一个支持多线程并发访问的 pickle 模块才可以完成这类任务。这时可以使用 ZODB。

ZODB 是一个健壮的、多用户的和面向对象的数据库系统，专门用于存储 Python 语言中的对象数据。它能够存储和管理任意复杂的 Python 对象，并支持事务操作和并发控制。ZODB 也是在 Python 的序列化操作基础之上实现的。想要有效地使用 ZODB，学好 pickle 是必须的。（ZODB 的使用不是本书重点，有兴趣的读者可以自行查阅 ZODB 的相关资料）

8.7 实例 24：批量读取及显示 CT 医疗影像数据

本案例取之一个真实的商业案例。案例中所用的代码源于 PACS Online 系统中的一部分。PACS Online 系统是北京中世康恺科技有限公司开发的医疗云影像 SaaS 系统。目前已经应用在国内 200 余家医疗机构中。

在本案例的实现中，使用了循环处理、异常处理、文件操作等知识，同时还使用了第三方库进行数据处理与图片生成。通过此案例的练习，可以增强读者 Python 编程的实战能力。

实例描述

编写一段代码，处理一组医疗影像数据，该数据为某用户 CT 扫描影像数据，扫描的位置为心脏器官。将该医疗影像里面的内容取出，并将该用户的全部 CT 影像数据转化成图片。

这是一个非常实际的应用案例。现在医疗影响行业所用的 CT 数据大多也都是 DICOM 格式。

8.7.1 DICOM 介绍

DICOM（Digital Imaging and Communications in Medicine，医学的数字成像和通信），是医学图像和相关信息的国际标准（ISO 12052）。它定义了可用于数据交换的医学图像格式，能够满足临床的需要。该医学图像格式可用于处理、存储、打印和传输医学影像信息。

DICOM 被广泛应用于放射医疗、心血管成像以及放射诊疗诊断设备（X 射线、CT、核磁

共振、超声等）所输出的格式，并且在眼科和牙科等医学领域得到越来越深入广泛的应用。在数以万计的在用医学成像设备中，DICOM 是被广泛部署的医疗信息标准之一。当前大约有百亿级符合 DICOM 标准的医学图像在临床使用。

在实际项目中，DICOM 是一个个以扩展名为“.dcm”的文件形式存在，如图 8-1 所示。

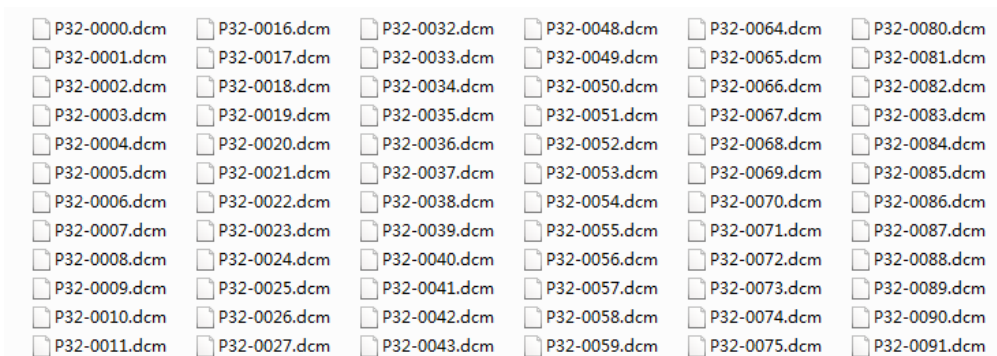


图 8-1 DICOM 文件

一个用户的 CT 影像数据，有可能包含上百张的 DICOM 文件。每个文件都是用户某个器官切片的截面影像数据。将所有的切面合起来，则构成了一个全面的用户身体内部影像信息。

8.7.2 Python 中的 DICOM 接口模块

在 Python 语言中，处理 DICOM 最常见的模块有 pydicom 与 dicom，都属于第三方模块，需要使用 pip 命令进行单独安装（安装方法参见第 2 章内容）。

本书将以 dicom 模块为例进行演示，所以，在编写代码前需要将 dicom 模块安装好。具体做法是，在命令行下输入如下命令：

```
pip install dicom
```

安装好后，就可以在代码中引入 dicom 模块，并调用 dicom 模块中的 read_file 函数来读取数据文件。

8.7.3 编写代码以载入 DICOM 文件

本案例的实验样本是一个用户的心脏器官 CT 数据。具体数据样本在随书的配套资源中“patient32”的文件夹里。将该文件夹复制到本地代码的同级路径下，就可以编写代码将其载入。

在代码中定义 `load_scan` 函数。在 `load_scan` 函数中，先使用 `dicom` 模块中的 `read_file` 函数将全部的数据读取到 `slices` 列表中。这里使用了列表推导式（见代码第 5 行），将指定路径下的数据依次读入并装到列表中。代码如下：

代码 8-4：处理医疗影像中的 DICOM 数据

```
01 import dicom                                #引入 dicom 模块
02 import os
03
04 def load_scan(path):                        #加载 DICOM 数据
05     slices = [dicom.read_file(path + '/' + s) for s in os.listdir(path)]#加载数据
06     return slices
07
08 INPUT_FOLDER = "patient32/P32dicom"         #指定文件路径
09 first_patient = load_scan(INPUT_FOLDER )    #调用函数，加载数据
10 print(first_patient[0])                    #将第一个数据打印
```

代码运行后，输出如下内容：

```
(0008, 0005) Specific Character Set          CS: 'ISO_IR 100'
(0008, 0008) Image Type      CS: ['DERIVED', 'SECONDARY', 'M', 'RETRO', 'DIS2D', 'CSA
RESAMPLED']
(0008, 0012) Instance Creation Date          DA: '20090312'
(0008, 0013) Instance Creation Time          TM: '085625.484000'
(0008, 0016) SOP Class UID                   UI: MR Image Storage
.....
(0040, 0253) Performed Procedure Step ID     SH: 'MR20090312084637'
(0040, 0254) Performed Procedure Step Descriptio LO: 'COEUR VAISSEaux VB13^COEUR'
(7fe0, 0010) Pixel Data                      OW: Array of 110592 bytes
```

上面的输出是 CT 影像中的一张 DICOM 文件内容。输出结果中的最后一行是 CT 影像的具体数据，而前面都是该数据的附加信息，如用户的年龄、名字及一些设备参数等。

8.7.4 读取 DICOM 中的数值

在得到 DICOM 数据之后，就可以调用该数据的 `dir` 方法来获得所有的属性，接着直接访问所需的属性，便可拿到具体的数值，代码如下：

代码 8-4：处理医疗影像中的 DICOM 数据（续）

```
11 print(first_patient[0].dir() )
12 print(first_patient[0].dir("pat") )
13 print(first_patient[0].PatientName )
```

第 2 行代码，在调用 `dir` 时传入了 `pat` 参数，意思是，在所有属性中输出属性名中含有 `pat` 字符串的属性。最后一行代码，是输出数据中的用户姓名。代码运行后输出如下结果：

```
[ 'AccessionNumber', 'AcquisitionDate', 'AcquisitionMatrix', 'AcquisitionNumber',
'AcquisitionTime', 'AngioFlag', 'BitsAllocated', 'BitsStored', 'CardiacNumberOfImages',
'Columns', 'ContentDate', 'ContentTime', 'DerivationDescription', 'DeviceSerialNumber',
'EchoNumbers', 'EchoTime', 'EchoTrainLength', 'FlipAngle', 'FrameOfReferenceUID',
'HeartRate', 'HighBit', 'ImageComments', 'ImageOrientationPatient',
'ImagePositionPatient', 'ImageType', 'ImagedNucleus', 'ImagesInAcquisition',
'ImagingFrequency', 'InPlanePhaseEncodingDirection', 'InstanceCreationDate',
'InstanceCreationTime', 'InstanceNumber', 'InstitutionAddress', 'InstitutionName',
'LargestImagePixelValue', 'LargestPixelValueInSeries', 'MRAcquisitionType',
'MagneticFieldStrength', 'Manufacturer', 'ManufacturerModelName', 'Modality',
'NominalInterval', 'NumberOfAverages', 'NumberOfPhaseEncodingSteps', 'PatientAge',
'PatientBirthDate', 'PatientID', 'PatientName', 'PatientPosition', 'PatientSex',
'PatientWeight', 'PercentPhaseFieldOfView', 'PercentSampling',
'PerformedProcedureStepDescription', 'PerformedProcedureStepID',
'PerformedProcedureStepStartDate', 'PerformedProcedureStepStartTime',
'PhotometricInterpretation', 'PixelBandwidth', 'PixelData', 'PixelRepresentation',
'PixelSpacing', 'PositionReferenceIndicator', 'ProtocolName',
'ReferringPhysicianName', 'RepetitionTime', 'Rows', 'SAR', 'SOPClassUID',
'SOPInstanceUID', 'SamplesPerPixel', 'ScanOptions', 'ScanningSequence', 'SequenceName',
'SequenceVariant', 'SeriesDate', 'SeriesDescription', 'SeriesInstanceUID',
'SeriesNumber', 'SeriesTime', 'SliceLocation', 'SliceThickness',
'SmallestImagePixelValue', 'SmallestPixelValueInSeries', 'SoftwareVersions',
'SpecificCharacterSet', 'StudyDate', 'StudyDescription', 'StudyID', 'StudyInstanceUID',
'StudyTime', 'TransmitCoilName', 'TriggerTime', 'VariableFlipAngleFlag',
'WindowCenter', 'WindowWidth', 'dBdt']

[ 'ImageOrientationPatient', 'ImagePositionPatient', 'PatientAge',
'PatientBirthDate', 'PatientID', 'PatientName', 'PatientPosition', 'PatientSex',
'PatientWeight']

60^DE
```

输出结果可以分为 3 段：前两个是列表，最后一行是字符串。

- 第一个列表是用户的全部属性。这些属性的内容构成了用户 CT 数据的全部信息。
- 第二个列表是部分属性名，可以看到每个里面都会有 `Pat` 字符串。
- 最后一行的字符串是用户的姓名，这里显示的是个特殊符号，原因是数据样本已经做了脱敏处理。

8.7.5 显示单张 DICOM 数据图像

下面引入图像显示模块 `pylab`，并使用简单的两行代码完成数据的显示。代码如下：

代码 8-4：处理医疗影像中的 DICOM 数据（续）

```
14 import pylab
15 pylab.imshow(first_patient[0].pixel_array, cmap=pylab.cm.bone)
16 pylab.show()
```

直接将单张 DICOM 数据的 `pixel_array` 属性传入图片显示函数 `pylab.imshow` 中，并调用 `pylab.show` 函数进行显示，输出结果如图 8-2 所示，显示的是一个心脏切片的截面图像。

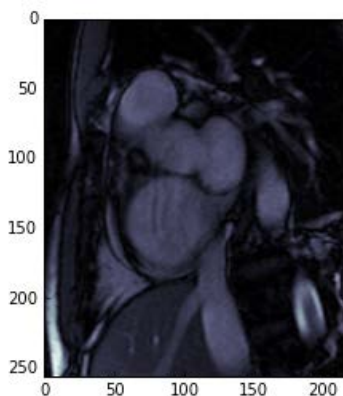


图 8-2 DICOM 单张图示

8.7.6 批量生成 DICOM 数据图像

案例的最后一步是生成一个该用户的全部 DICOM 图像数据。这里使用了另一个图片处理模块——PIL 下的 `Image`，该模块对图像的处理功能更为强大。本例中将使用该模块实现对所有图片的拼接，生成一张图片。

在代码中，定义了 `plot_ct_scan` 函数，以实现对 DICOM 数据的图像拼接。首先统计全部图片的个数，然后计算每个图片所占的尺寸，最后使用 `paste` 函数将图片拼接在一起，并调用 `save` 函数保存起来。代码如下：

代码 8-4：处理医疗影像中的 DICOM 数据（续）

```
17 import PIL.Image as Image
18 import math
```

```

19
20 def plot_ct_scan(scan):
21     length = len(scan)                                #获取文件夹内的文件个数
22     each_size = int(math.sqrt(float(810*810)/length)) #根据总面积求出每一个图片的大小
23     lines = int(810/each_size)                         #算出每一行可以放多少个图片
24     image = Image.new('RGBA', (810, 810), 'white')    #生成白色背景新图片
25     x = 0
26     y = 0
27     for i in range(0,length):                          #依次生成图片
28         img=Image.fromarray(scan[i].pixel_array.astype(int)) #生成img 对象
29         img = img.resize((each_size, each_size), Image.ANTIALIAS) #调整图片大小
30         image.paste(img, (x * each_size, y * each_size)) #拼接图片
31         x += 1
32         if x == lines:
33             x = 0
34             y += 1
35     image.save('./' + "all.jpg")                        #保存图片
36
37 plot_ct_scan(first_patient)                            #调用函数

```

全部代码执行后，在本地代码的同级目录下可以看到有一张 all.jpg 的图片，如图 8-3 所示。

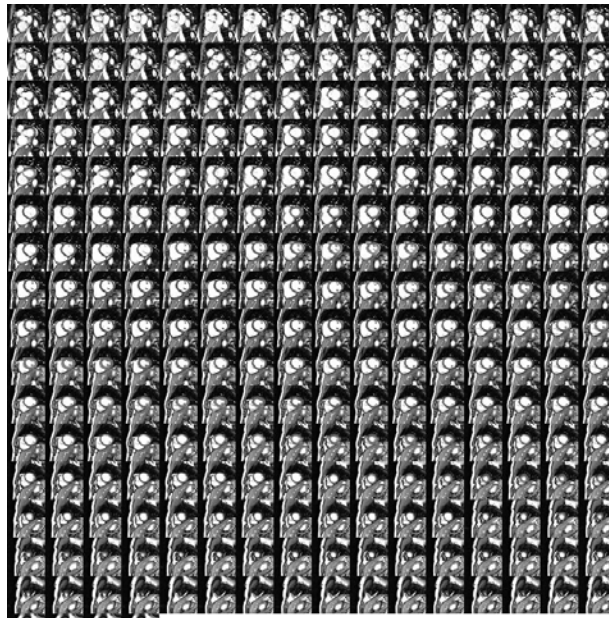


图 8-3 显示全部 DICOM 数据

第 3 篇 高阶

本篇介绍了 Python 语言在面向对象编程及系统调度方面的知识。使用这些高级技术，能够开发出更加高效、稳定、易扩展的代码。

- ▶ 第 9 章 类——面向对象的编程方案
- ▶ 第 10 章 系统调度——实现高并发的处理任务

第 9 章

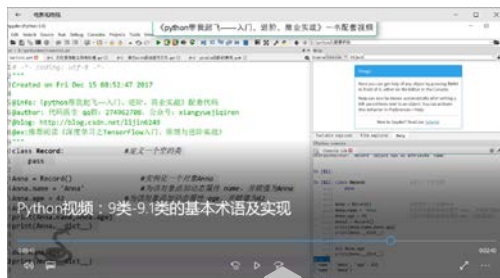
类——面向对象的编程方案

Python 是一种面向对象的脚本语言。面向对象是一种编程思想，其含意是：将功能封装进对中，通过对象方法去实现具体的细节。

面向对象的编程思想更符合人们的思考习惯，可以将复杂的事情简单化。在开发过程中，面向对象编程思想会从数据出发，将数据抽象为对象，把要实现的功能定位为对象的方法。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解

具体代码操作演示



Python视频：
9类-9.1类的基本术语及实现.mp4



Python视频：
9类-9.2实例化对象.mp4



Python视频：
9类-9.3私有化.mp4



Python视频：
9类-9.4子类.mp4



Python视频：
9类-9.5内置函数9.6重载运算符.mp4



Python视频：
9类-9.7包装与代理.mp4



Python视频：
9类-9.8自定义异常类.mp4



Python视频：
9类-9.9自定义with类9.10自定义迭代器9...

本章共有 8 段教学视频，总时长为 58 min 左右。包含下内容：

- 讲述类的相关术语及实现方法。
- 讲述实例化类对象。
- 讲述类变量的私有化类属性。
- 讲述实现子类。
- 讲述类相关的常用内置函数和重载运算符。
- 讲述包装与代理。
- 讲述自定义异常类。
- 讲述支持 with 语法的自定义类、自定义迭代器类的实现、调试技巧和元类。

9.1 类的相关术语及实现

Python 中使用类（class）来实现面向对象编程。Python 中的类，具有面向对象编程的所有基本特征：允许多继承、派生类可以重写它父类的任何方法、方法可以调用父类中同名的方法，对象可以包含任意数量和类型的数据成员。

1. 类的相关术语

在 Python 中，类相关的面向对象术语如下：

- 类（class）：用来描述具有相同的属性和方法的对象的集合。它定义了该集合中所有对象共有的属性和方法。
- 类变量：类变量在所有实例化对象中是公用的。类变量定义在类中，且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量，用于处理类及其实例对象的相关数据。
- 方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称作方法的重写。
- 实例变量：定义在方法中的变量，只作用于当前实例的类。
- 继承：即派生类（derived class）具有基类（base class）一样的字段和方法。继承会把一个派生类的对象作为一个基类对象对待。
- 实例化：创建一个类的实例，即得到类的具体对象。
- 方法：类中所定义的函数，也叫作该类的成员函数。
- 对象：对象是类的实例。对象包括两个数据成员（类变量和实例变量）和方法。

9.1.1 创建类

Python 中，使用 `class` 语句来创建一个新类。`class` 之后为类的名称，并以冒号结尾。最简单的类的定义如下：

```
class ClassName:
    <语句-1>
    .....
    <语句-N>
```

在 `class` 的代码块中，可以编写该类的执行语句。一般会先使用多行字符串对该类进行说明，这一点与函数类似。

类定义会创建一个新的命名空间作为一个局部的作用域。在该作用域中的变量被叫作成员变量。在该类作用域下的函数就被叫作成员函数，也叫作该类的方法。当一个类定义结束后，系统就会创建一个类对象（Class Object）。

9.1.2 创建类属性

下面创建一个类，并定义该类的说明字符串、成员变量、方法。

1. 定义类属性

需要注意的是，在定义类方法时，需要有一个默认的形参——`self`。而在调用类方法时，却不需要传入形参 `self` 所对应的实参。因为这个形参 `self` 是由类的内部机制调用的。具体如下：

```
class MyClass:                                #定义一个类
    """A simple example class"""              #定义该类的说明字符串
    i = 12345                                  #定义成员变量
    def f(self):                               #定义方法
        return 'I love Python'
```

2. 使用类

创建好的类可以实例化成一个对象，并通过调用对象的方法来实现具体功能。例如：

```
myc = MyClass ()                             #实例化类对象，并赋值给 myc
print(myc.i)                                 #将类实例 myc 的成员变量 i 打印。输出：12345
print(myc.f())                               #调用类实例 myc 的成员函数 f，并打印返回值。输出：I love Python
```

自定义的类与 Python 中的内置类型，在使用上几乎一样。生成 `MyClass` 实例化的对象 `myc`

后，就可以调用 myc 中的变量 i 和方法 f 了。

3. 类的内置属性

Python 中的类有一些相同的内置属性。这些内置属性用于维护类的基本信息，具体如下。

- `__name__`：类名称。
- `__doc__`：类的文档字符串。
- `__module__`：类定义所在的模块。如果是直接运行当前文件，该值为 `__main__`。
- `__base__`：该类所有的父类 `<class 'object'>` 列表，是一个 `tuple` 类型的对象。
- `__dict__`：该类的所有属性（由类的数据属性组成），是一个 `dict` 类型的对象。

将 MyClass 类的内置属性打印出来，代码如下：

```
print(MyClass.__name__)          #打印类的名字，输出：MyClass
print(MyClass.__doc__)           #打印类的文档字符串，输出：A simple example class
print(MyClass.__module__)        #打印类定义所在的模块，输出：__main__
print(MyClass.__base__)          #打印类的所有父类，输出：<class 'object'>
print(MyClass.__dict__)          #打印类的属性，输出：{'f': <function MyClass.f at 0x000000000C499048>, '__dict__':
<attribute '__dict__' of 'MyClass' objects>, '__weakref__': <attribute '__weakref__'
of 'MyClass' objects>, 'i': 12345, '__doc__': 'A simple example class', '__module__':
'__main__'}
```

9.1.3 定义类的动态属性

Python 中的类，在使用时能够根据需要在需要时随时添加动态属性。这一特性会使编码变得更简单。

1. 添加动态属性

为类添加动态属性的方法是：在实例化对象的后面加一个点，在点后跟上属性名称。

如果该对象已经有这个属性，系统就会读取其内容；否则，系统会自动为该对象加上这个属。

例如下面代码，定义了一个空的类，并为空类动态添加属性：

```
class Record:                    #定义一个空的类
    pass

Anna = Record()                 #实例化一个对象 Anna
Anna.name = 'Anna'              #为该对象添加动态属性 name，并赋值为 Anna
```

```
Anna.age = 42                #为该对象添加动态属性 age，并赋值为 42

print(Anna.name,Anna.age)    #将该对象的 name 和 age 属性打印。输出：Anna 42
```

上面代码中定义了一个空的类 **Record**。类中的 `pass` 语句不做任何操作，只是占一个位置而已（见前面 5.4 节）。接着用类 **Record** 实例化一个对象 **Anna**。为 **Anna** 动态添加了两个属性（`name` 和 `age`），并为它们赋值。最后尝试访问这两个属性，得到了属性对应的值。

这是一种非常高效的开发方式。读取文件、处理已知或未知结构体，都可以使用该方法。通过对一个类进行实例化，就可以生成一个与目标数据具有一样结构的对象，可以对其任意写入和读取。



注意：

这里使用了一个空类来举例，是因为在处理动态属性问题时，使用空类的做法比较常见。实际规则中，所有的类都具有这种动态添加属性的特性。

2. 删除动态属性

虽然可以动态地添加属性，但总不能无止境地增加很多属性。在有时还需将不用的属性删除。用 `del` 加上对象属性，即可删除属性。例如，接上面的例子添加如下代码：

```
del Anna.age                #删除 Anna 对象的属性 age
print(Anna.age)             #再次访问 Anna 的 age 属性，会报错
```

上面的代码中，使用了 `del` 语句将 `age` 属性从 **Anna** 对象中删除。再次访问 **Anna** 的 `age` 属性时，就会报如下错误：

```
AttributeError: 'Record' object has no attribute 'age'
```

这表明，对象 **Anna** 中已经没有 `age` 属性了。

9.1.4 限制类属性（`__slots__`）

在 9.1.3 小节中介绍过，可以为实例化后的类对象任意添加动态属性。这一特性是一把双刃剑，在提升了代码灵活性的同时，也容易使类对象失去控制。

在团队合作过程中，当某些底层类需要被其他接口调用时，为了保证接口的可控性，往往希望调用者按照规定的情况来使用。即，将类的属性固定下来。这种情况下，可以在类中对 `__slots__` 变量赋值，从而限制类的属性。

`__slots__`是一个特殊的变量，其值可以是元组的形式，元组的元素为该类所允许添加的属性名称。在一个类中，一旦为`__slots__`赋值，则该类的实例化对象就只能添加`__slots__`中所规定的属性。例如：

```
class Record:                                #定义一个空的类
    __slots__ = ('name', 'age')              #设置合法的属性为 name 与 age

Anna = Record()                              #实例化一个对象 Anna
Anna.name = 'Anna'                          #为该对象添加动态属性 name，并赋值为 Anna
Anna.age = 42                               #为该对象添加动态属性 age，并赋值为 42
Anna.age2 = 42                               #系统报错，AttributeError: 'Record' object has no attribute 'age2'
```

上面代码中，在 `Record` 类中为 `__slots__` 变量添加了合法的属性 `name` 和 `age`。于是在最后一行为 `Record` 实例化对象 `Anna` 添加 `age2` 时，报了错误，提示 `Record` 没有 `age2` 这个属性。



注意：

特殊变量 `__slots__` 在类的派生过程中仍然有效，即，子类会继承父类的 `__slots__` 内容。关于继承与派生的知识可以参考 9.4 节。

9.2 实例化类对象

前面的 9.1.2 小节中“2. 使用类”中，介绍了类实例化的基本用法。这里介绍更多的用法。

9.2.1 带有初始值的实例化

实例化类，可以使用类的名字加括号来实现。对于一些复杂功能的类，还需要在实例化的同时为其初始化某些必需的成员变量，这就是带有初始值的实例化。

1. 类的初始化方法（`__init__`）

要想实现带有初始值的实例化，需要在定义类时，在类里面实现一个 `__init__` 方法。`__init__` 方法的定义与函数几乎一样，也需要形参，并且支持形参默认值等规则。这样，在实例化时，可以将具体要初始化的值当作实参，传入到类名字后的括号里。例如：

```
class MyClass:                                #定义一个类
    """A record class"""                    #定义该类的说明字符串
    def __init__(self, name, age):          #定义该类的初始化函数
        self.name = name                    #将传入的参数值赋值给成员变量
        self.age = age
```

```

def getrecode(self):          #定义一个成员函数
    return self.name,self.age #该成员函数返回该类的成员变量

myc =MyClass ("Anna",42)      #实例化一个对象，并为其初始化
print(myc.getrecode())        #调用对象的成员函数，并将返回值打印。输出元组类型：('Anna', 42)

```

在初始化函数`__init__`中，直接使用了`self`加点的方式，为指定类中的成员变量赋值。这个成员变量的定义机制与普通变量的定义机制类似，在赋值语句执行的同时，会自动创建并加入该类。

2. 隐藏调用类的初始化方法

其实每个类在实例化时，都会在内部调用初始化函数`__init__`。对于一个没有初始化函数的类，在实例化时，也会调用内部默认的`__init__`函数。如果在类中实现了函数`__init__`，就会优先调用自定义的函数`__init__`。例如：

```

class MyClass:                #定义一个类
    """A record class"""      #定义该类的说明字符串
    def __init__(self):        #定义该类的初始化函数
        print("here")         #将传入的参数值赋值给成员变量

myc =MyClass ()               #实例化一个对象。输出：here

```

上面代码中，在类 `MyClass` 内部建立了一个`__init__`函数，`__init__`的函数体是输出一句话。在对 `MyClass` 进行实例化时可以看到，屏幕上自动输出了 `here`。这表明，实例化类时，即使没有初始值也会调用`__init__`函数。



注意：

如果类中的`__init__`函数，有除 `self` 以外的参数。实例化该类时，就必须输入与`__init__`函数对应的参数，否则就会报错。

9.2.2 class 中的 self

在类的代码块中，可以使用 `self` 关键字来指定本身。在前面 9.1.2 小节的“1. 定义类属性”中已经讲过，类中的成员函数必须第一个参数是 `self`。这样才可以保证，使用类实例化的对象能够调用自己的成员函数。

1. 直接调用类的成员函数

从函数传值的角度来看，直接使用该类的成员函数并传入实例化对象，与通过“类对象.成员函数”的调用方法效果一样。例如：

```
class MyClass:                                #定义一个类
    """A simple example class"""             #定义该类的说明字符串
    i = 12345                                 #定义成员变量
    def f(self):                              #定义方法
        return 'I love Python'

myc =MyClass ()                               #实例化类对象，并赋值给 myc
print(myc.f())                               #调用类实例 myc 的成员函数 f，并打印返回值。输出: I love Python
print(MyClass.f(myc))                       #调用类实例 myc 的成员函数 f，并打印返回值。输出: I love Python
```

上面代码中显示定义了一个类 `MyClass`，并在类 `MyClass` 中定义了 `f` 方法。

最后两句，分别使用了“实例化对象（`myc`）+点+函数（`f`）”和“向类（`MyClass`）的函数（`f`）中传入对象（`myc`）”的方式来调用函数。

可以看到，它们的效果是一样的。

2. 将成员函数定义在类外面

如果将类中的成员函数（即方法）看成一个函数，则完全可以把类的成员函数定义在类外面。但是，第一个参数是 `self` 这个规则还要必须保留。例如：

```
def fun(self):                                #在类的外面定义方法
    return 'I love Python'

class MyClass:                                #定义一个类
    """A simple example class"""             #定义该类的说明字符串
    i = 12345                                 #定义成员变量
    f = fun                                   #在类的内部指定成员函数

myc =MyClass ()                               #实例化类对象，并赋值给 myc
print(myc.f())                               #调用类实例 myc 的成员函数 f，并打印返回值。输出: I love Python
```

可以看到，函数 `fun` 定义在类 `MyClass` 的外面，但是不影响整个程序的功能。通过在类中对 `fun` 的指定，仍然可以访问到。

这种方法使得成员函数可以被多个类使用，从而提高了代码的重用性。

3. 类内的成员互访

如想在类中使用该类的其他成员变量或方法，需要使用 `self` 加点的方式来实现。例如：

```
class MyClass:                                #定义一个类
    """A record class"""                     #定义该类的说明字符串

    def __init__(self, name, age):            #定义该类的初始化函数
        self.name = name                     #将传入的参数值赋值给成员变量
        self.age = age

    def getage(self):                          #定义一个成员函数，返回 age 值
        return self.age

    def getname(self):                        #定义一个成员函数，返回 name 值
        return self.name

    def getrecode(self):                      #定义一个成员函数，返回 age 与 name 的值
        return self.getage(),self.getname()

myc =MyClass ("Anna",42)                      #实例化一个对象，并为其初始化
print(myc.getrecode())                       #调用对象的成员函数，并将返回值打印。输出元组：('Anna', 42)
```

在上面的例子中，类 `MyClass` 中的方法调用了同属于该类中的 `getage` 和 `getname` 方法。所以在这两个方法前面加上 `self` 和点。类似的，在 `getage` 方法中要返回该类的成员变量 `age`，同样也要使用 `self` 加点的方式才可以访问。



注意：

在 Python 中，每个值都是一个对象，可以通过 `object.__class__` 来获取对象的 `class`（即类型），其作用与 `type()` 相同。

9.2.3 类方法（@classmethod）与静态方法（@staticmethod）

在 9.2.2 小节中讲到了 `class` 中的 `self`，指的是类实例本身。在默认的情况下，类的成员方法（也叫成员函数）的第一个参数必须是 `self`。在类中还有两种其他类型的成员函数：类方法与静态方法。

1. 类方法

在一个类的定义中，如某个方法使用了装饰器 `@classmethod` 进行装饰，则说明该方法是一个类方法。

类方法与默认成员方法的区别是：类方法属于类；而默认成员方法属于类实例化对象。具

体表现为:

- 类方法的第一个参数必须是 `cls` (用来指代该类), 而默认成员方法的第一个参数必须是 `self` (用来指代该类的实例对象)。在调用类方法时, 需要使用“类名.类方法名”的方式。
- 在调用默认成员方法时, 则是使用“该类的实例化对象.类方法名”的方式。

具体代码如下:

```
class MyClass:                                #定义一个类
    def f(self):                               #默认的成员方法
        return 'I love Python'

    @classmethod                               #定义类方法
    def fcls(cls):
        return '类方法: I love Python'

myc =MyClass ()                               #实例化类对象, 并赋值给 myc
print(myc.f())                                #调用类实例 myc 的成员函数 f, 并打印返回值。输出: I love Python
print(MyClass.fcls())                         #调用类方法, 输出 “类方法: I love Python”
```

上面代码中, `fcls` 为类方法, 使用了装饰器 `@classmethod` 进行装饰。在调用时, 直接使用类名 `MyClass` 进行调用。在最后一行代码中, 将类方法 `fcls` 返回的字符串成功地打印出。

2. 静态方法

类似于类方法, 静态方法的定义是使用装饰器 `@staticmethod` 进行装饰的。静态方法的特点是: 第一个参数没有任何要求, 与普通的函数参数一样。其调用方式更加宽松, 使用类名或是类的实例化对象都可以对其进行调用。代码如下:

```
class MyClass:                                #定义一个类
    @staticmethod                              #定义静态方法
    def fstatic( ):
        return '静态方法: I love Python'

myc =MyClass ()                               #实例化类对象, 并赋值给 myc
print(myc.fstatic())                          #调用类方法, 输出 “静态方法: I love Python”
print(MyClass.fstatic())                      #调用类方法, 输出 “静态方法: I love Python”
```

上例中, `MyClass` 类中定义了成员函数 `fstatic`, 并且用装饰器 `@classmethod` 进行装饰。这表明 `fstatic` 是静态方法。代码的最后两行, 分别使用了类实例化对象与类名对其进行调用。可

以看到，程序能够正常执行，并且成功地将类方法 `fstatic` 所返回的字符串打印出。

3. 总结

类成员方法有三种类型，分别面向于类中的三个实体。

- 默认的成员方法：仅仅面向该类的一个实例。
- 类方法：面对的是当前类，独立于类实例化的各个对象。
- 静态方法：等同于普通函数，只是被封装在类中而已，其独立于整个类。



注意：

类成员方法的三种类型，在类的派生过程中仍然有效。即，子类也可以调用父类的成员方法。关于继承与派生的相关知识可以参考 9.4 节。

9.2.4 类变量与实例变量的区别

类变量与实例变量的区别如下。

- 类变量：是指类的属性和方法，类似于一种静态数据。类变量是在定义类时所定义的，类的所有实例都会共享类变量；
- 实例变量：更像是一种动态数据。只有在实例化时，系统才会为该实例指定它特有的数据，有别于该类中的其他个体。

1. 类变量与实例变量举例

例如 9.2.1 小节的例子中，`MyClass` 为一个记录类，用来放置人物的记录。在设计类结构时，也可以继续细分，比如让 `MyClass` 代表一类“职业为科学家的人”的记录。可以在 `MyClass` 在里面加一个职业的属性。具体代码如下：

```
class MyClass:                                #定义一个类
    """A record class"""                     #定义该类的说明字符串
    Occupation = "scientist"                  #职业为科学家
    def __init__(self, name, age):           #定义该类的初始化函数
        self.name = name                     #将传入的参数值赋值给成员变量
        self.age = age

    def getrecode(self):                      #定义一个成员函数
        return self.name,self.age           #该成员函数返回该类的成员变量
```

代码中，变量 `Occupation` 在类中定义，属于类变量；而 `name` 与 `age` 是以 `self` 开头，属于实例变量。

当用该类实例化一个人物 `Anna` 时，`Anna` 则为该类特有的个体。

```
myc = MyClass ("Anna", 42)           #实例化一个对象，并为其初始化
```

`Anna` 为 `MyClass` 的一个个体，她有自己单独的名字“`Anna`”和年纪“42”。但她也有 `MyClass` 类的共用属性，即“职业为科学家”。

也可以使用类似方法造出更多的该类个体，例如“`Gary`”等。无论起什么名字，从 `MyClass` 实例化出来的个体，都具有“职业为科学家”的属性。

2. 类变量与实例变量的易犯错误

通过修改一个类的类变量，可以将这个类的所有实例化对象的属性统一修改。但是，当实例变量与类变量同名时，系统会以实例变量优先。这会导致类变量失效，从而无法批量修改实例化后的类属性。

例如，接上面的代码，再实例化一个个体 `Gary`，将 `Gary` 的职业属性改成了“发明家”，却不小心将 `Anna` 的职业也改变了。代码如下：

```
myc2 = MyClass ("Gary", 38)           #实例化一个对象 Gary，并为其初始化
myc2.Occupation = "inventor"          #将 Gary 的职业属性改成了发明家
MyClass.Occupation = "dancer"         #将职业属性批量修改成了舞蹈家
print(myc.Occupation)                 #打印出 Anna 的职业。输出：dancer
print(myc2.Occupation)                #打印出 Gary 的职业。输出：inventor
```

上面代码就是一个错误的例子，程序执行的结果与开发者的意图相违背。这是值得注意的地方。在开发时，一定要分清哪个是类的真正“个体数据”，而哪个是类的“共享变量数据”。如果让“个体数据”与“共享变量数据”同名，将会出现意想不到的错误。

9.2.5 销毁类实例化对象

面向对象编程思想中，每个类的实例化对象都是一个完整的个体。即，类可以自己进行初始化和销毁。

在创建类实例化对象时，可使用内置的 `__init__` 初始化该对象；当该对象需要销毁时，也需要进行一次自我销毁操作。调用 `__del__` 方法可以销毁对象，类的 `__del__` 方法与初始化方法

`__init__`是一一对应的。例如：

```
class MyClass:                                #定义一个类
    """A record class"""                     #定义该类的说明字符串

    def __init__(self, name, age):            #定义该类的初始化函数
        self.name = name                     #将传入的参数值赋值给成员变量
        self.age = age

    def __del__(self):                         #销毁该类的实例化对象
        print(self.__class__.__name__, "del")

myc =MyClass ("Anna", 42)                     #实例化一个对象，并为其初始化
del myc                                       #删除该实例对象，输出：MyClass del
```

上面的代码，在执行最后一句时，输出了 `MyClass del`。这表明在删除 `myc` 实例时，程序调用了 `__del__` 函数。

`__del__` 与 `__init__` 都是与类操作有关的内置函数。当一个类中没有重写该函数时，程序什么都不做。重写之后，在创建实例化对象时，会自动调用 `__init__` 函数。在销毁对象时，会自动调用 `__del__` 函数。

9.3 类变量的私有化类属性

前面 9.2.4 小节中“2. 类变量易犯错误”的例子，分析了类变量容易读取失败的原因：类变量与实例变量重名。

为了避免这个问题，可以给类变量加上一定的权限，不允许实例化后的对象直接访问类变量。通过将类变量隐藏起来，避免读取类变量失效的问题。这就是类的私有化的初衷。

9.3.1 公有化（public）与私有化（private）

私有化的实现方式是：给类变量加上一个私有化（`private`）权限。被私有化的属性不能被该类的实例化对象直接访问，但是类的内部成员函数是可以访问的。如果类的实例化对象想要取得该类的私有化属性，可以通过调用该类中的函数来完成。

与私有化（`private`）对应的是公有化（`public`）。公有化是 Python 中的默认权限，即，所有的对象都可以访问。

9.3.2 私有化的实现

在 Python 中规定，以两个下划线开头的成员对象为私有化成员。

1. 私有化的实现举例

下面通过私有化类属性，来避免类属性被任意篡改。代码如下：

```
class MyClass:                                #定义一个类
    """A record class"""                     #定义该类的说明字符串
    __Occupation = "scientist"                 #职业为科学家，私有变量
    def __init__(self, name, age):            #定义该类的初始化函数
        self.name = name                     #将传入的参数值赋值给成员变量
        self.age = age

    def getrecode(self):                      #定义一个成员函数
        return self.name,self.age            #该成员函数返回该类的成员变量

    def getOccupation(self):                  #返回私有变量的方法
        return self.__Occupation
```

上面的代码是由 9.2.4 小节中“2. 类变量易犯错误”的例子修改而来。

可以看到，代表职业的类属性名字前面加了两个下划线，变为“__Occupation”，这代表私有属性。同时又提供了该私有化属性“__Occupation”的访问函数 getOccupation，会返回该类的职业。

2. 使用私有化方法保护实例对象

接下来，实例化两个对象，并修改其中一个对象的职业，看看会发生什么。代码如下：

```
myc =MyClass ("Anna",42)                     #实例化一个对象 Anna，并为其初始化
myc2 =MyClass ("Gary",38)                    #实例化一个对象 Gary，并为其初始化
myc2.__Occupation = "inventor"                #将 Gary 的职业属性改成了发明家
print(myc.getOccupation())                    #打印出 Anna 的职业。输出：scientist
```

上面的代码实例化了两个对象，Anna 与 Gary。将 Gary 的职业属性改成了发明家(inventor)，并调用 Anna 对象的 getOccupation 方法来获得职业。

运行上述代码后可以看到，程序输出了 Anna 的职业是科学家（scientist）。这表明，修改 Gary 的职业属性并没有影响到 Anna。

3. 演示私有化的不可见性

接下来，分别打印 Gary 的 `__Occupation` 属性、`getOccupation` 方法的返回值，观察 Gary 的职业如何变化。代码如下：

```
print(myc2.__Occupation)           #打印出 Gary 的__Occupation。输出: inventor
print(myc2.getOccupation())        #打印出 Gary 的职业。输出: scientist
```

可以看出，Gary 的 `__Occupation` 属性（inventor）与 `getOccupation` 方法的返回值（scientist）并不一样。这表明，`myc2` 的 `__Occupation` 与 `MyClass` 类无关，是 `myc2` 对象自己添加的一个属性。而类 `MyClass` 中的 `__Occupation` 值并没有变化，只是 `myc2` 对象访问不到而已。只有调用该类的 `getOccupation` 方法才可返回该类中私有属性 `__Occupation` 的值：scientist。

打印 Anna 的对象（`myc`）的 `__Occupation` 属性。代码如下：

```
print(myc.__Occupation)           #系统报错，AttributeError: 'MyClass' object has no
attribute '__Occupation'
```

执行上面的代码会报错，提示 `MyClass` 没有 `__Occupation` 属性。再次证明了，私有变量 `__Occupation` 对于实例化对象是不可见的。



注意：

一般来讲，一个私有属性会有两个方法与之对应，即，`get` 与 `set`。`Get` 方法用来获取该私有属性的值，`set` 方法用来修改私有属性的值。

4. 私有化的原理

在 Python 中，私有化的原理是通过将私有化变量改名的方式实现的。可以看下面代码：

```
print(MyClass.__dict__)           #输出 MyClass 的类属性
```

这句代码是要输出 `MyClass` 的类属性。运行之后可以看到如下结果：

```
{'__module__': '__main__', '__doc__': 'A record class', '_MyClass__Occupation':
'scientist', '__init__': <function MyClass.__init__ at 0x00000260ED4127B8>, 'getrecode':
<function MyClass.getrecode at 0x00000260ED4122F0>, 'getOccupation': <function
MyClass.getOccupation at 0x00000260ED412378>, 'SetOccupation': <function
MyClass.SetOccupation at 0x00000260ED412840>, '__dict__': <attribute '__dict__' of
'MyClass' objects>, '__weakref__': <attribute '__weakref__' of 'MyClass' objects>}
```

可以看到显示的类属性中，有一个 `_MyClass__Occupation` 变量。这个变量就是私有变量 `__Occupation`。这表明 Python 语法会自动将私有变量改名字，通过这种方式，让实例化对象找

不到该私有变量。

当然，也可以直接访问这个名字来验证结果。代码如下：

```
print(myc._MyClass__Occupation) #输出了私有变量的值: scientist
```

这表明其实私有变量也是可以访问的。只是改了个名字而已。这部分知识是方便读者更深层的了解私有化原理。在实际编程时可以借鉴。

9.3.3 使用装饰器技术实现类的私有化（@property）

在 9.3.2 小节的例子中，私有化属性的应用提高了程序的健壮性。但必须为私有化变量封装 get 方法，通过调用 get 方法才可以取值。

由于私有化属性不能被直接访问，这使得类属性的取值方式发生了改变。下面介绍一种装饰器，它可将 get 或 set 方法装饰成类中的一个属性，从而使私有化的类属性也可以被直接访问。

1. 装饰器实现私有变量的 get 函数

对于 9.3.2 小节的例子，私有变量__Occupation 的 get 函数为 getOccupation。使用装饰器方式可以写成如下：

```
class MyClass:
    """A record class"""
    __Occupation = "scientist"
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def getrecode(self):
        return self.name,self.age

    def getOccupation(self):
        return self.__Occupation

    @property
    def Occupation(self):
        return self.__Occupation

myc =MyClass ("Anna",42)
print(myc.getOccupation())
print(myc.Occupation)
```

#定义一个类
#定义该类的说明字符串
#职业为科学家，私有变量
#定义该类的初始化函数
#将传入的参数值赋值给成员变量
#定义一个成员函数
#该成员函数返回该类的成员变量
#返回私有变量的方法
#将 get 函数装饰成属性
#实例化一个对象 Anna，并为其初始化
#打印出 Anna 的职业。输出: scientist
#以属性的访问方式来调用装饰器的函数。输出: scientist

上面的代码中，MyClass 的实现比 9.3.2 小节例子中的 MyClass 多了一个 Occupation 函数。

在 Occupation 函数的上方添加装饰器 @property，就可以直接把 Occupation 当成属性来用（见上面代码的最后一行）。程序运行后，调用 getOccupation 函数与直接访问 Occupation 输出了同样的结果：scientist。

2. 装饰器实现私有变量的 set 函数

继续扩展上面“1. 装饰器实现私有变量的 get 函数”的例子，对变量 __Occupation 的值做业务逻辑方面的限制，即，只允许变量 __Occupation 的值为 inventor 或 scientist。若改成其他值，让程序报错。实现时，可以在 MyClass 类中添加一个 __Occupation 的 set 函数。代码如下：

```
class MyClass:                                #定义一个类
    """A record class"""                     #定义该类的说明字符串
    __Occupation = "scientist"                 #职业为科学家，私有变量
    def __init__(self, name, age):             #定义该类的初始化函数
        self.name = name                     #将传入的参数值赋值给成员变量
        self.age = age

    @property
    def Occupation(self):                     #将 get 函数装饰成属性
        return self.__Occupation

    @Occupation.setter
    def Occupation(self,value):               #将 set 函数装饰成属性
        if (value != 'inventor') and (value != 'scientist'):
            raise (ValueError('Occupation must be inventor or scientist!'))
        else:
            self.__Occupation = value

myc =MyClass ("Anna",42)                      #实例化一个对象 Anna，并为其初始化
print(myc.Occupation)                         #打印出 Anna 的职业。输出：scientist
myc.Occupation = 'inventor'                   #以属性的访问方式来调用装饰器的函数。进行私有变量修改
print(myc.Occupation)                         #打印出 Anna 的职业。输出：inventor
```

上面代码是在原有的基础上，增加了一个被 @Occupation.setter 装饰的 Occupation 函数，在 Occupation 函数里，对传入的参数进行判断。如果传入的参数既不等于 inventor 又不等于 scientist，就抛出异常；否则就修改私有变量 __Occupation 的值。

很明显，函数 Occupation 是私有变量 __Occupation 的 set 函数。通过 @Occupation.setter 装饰之后，就可以当作属性被调用了（代码中倒数第二行）。

最后一行打印 Anna 职业时，输出的值为 `inventor`，这表明倒数第二行的代码已经生效，已成功修改了私有变量 `__Occupation` 的值。

同样，在 `@Occupation.setter` 装饰的 `Occupation` 函数中，对参数的检查也是生效的。

接着上面代码，添加如下代码：

```
myc.Occupation = 'doctor'           #抛出异常。ValueError: Occupation must be inventor or
scientist!
```

执行后，程序会报出异常。提示修改 `Occupation` 的值非法。

使用“装饰器+私有变量”的方式，在提升代码健壮性的同时，又保留了对属性直接访问的取值方式，使得 `get` 与 `set` 的操作变得透明。这样封装出的类，使用起来会更加安全和方便。

9.4 实现子类

在面向对象编程思想中，类与类之间可以有派生或是继承的关系。派生与继承是针对与父类、子类的关系而言。

- 父类可以派生一个子类。这样，子类也就继承了父类的属性与方法。
- 子类里也同样可以定义自己的属性与方法，并且能够派生出新的子类。

通过这种父/子类的编程思想来设计架构，是典型的面向对象思想。它可以用类的方式把复杂的需求抽象出来。即，把所有要描述对象的共性总结出来：对于全部都遵守的共性，用父类来描述；对于满足部分共性的对象，用多个子类来描述。

9.4.1 继承

类的继承分为单继承和多继承。

- 单继承是指派生类只有一个父类。即，只继承了一个父类的属性及方法。
- 多继承是指，派生类有多个父类。即，子类继承了多个父类的属性及方法。

1. 单继承的实现

单继承的实现非常简单。定义类时，在类名后面加一个括号，在括号里指定父类的类名。具体形式如下：

```
class DerivedClassName(FatherClassName):
```



```
<语句-1>
.....
<语句-N>
```

父类名 `FatherClassName` 对于派生类来说，必须是可见的。

也可以继承在其他模块中定义的基类。例如：

```
class DerivedClassName(module.FatherClassName):
```

当访问派生类的属性时，首先会在当前的派生类中搜索，如果没有找到，则会递归地去基类中寻找。

2. 子类方法覆写（override）

当子类的方法与父类的方法同名时，父类的方法将失效。即，子类方法覆写（override）了父类的方法。比较常见的覆写方式是：在子类里，执行自己的覆写方法的同时，也要调用一下父类的被覆写的方法。这种情况，可以通过以下方式直接调用父类方法：

```
FatherClassName.method(self, arguments)
```

3. 多重继承的实现

多重继承与单继承类似，只不过是类名后的括号里多加几个父类，中间用逗号分割。具体形式：

```
class DerivedClassName(FatherClassName1, FatherClassName2, .....):
<语句-1>
.....
<语句-N>
```

在多重继承下，若要访问派生类的属性，默认的搜索的规则是：深度优先，从左到右。即：

- （1）如果一个属性在当前类中没有被找到，它就会搜寻 `FatherClassName1`。
- （2）如果 `FatherClassName1` 中没找到，就会递归地搜寻 `FatherClassName1` 的父类。
- （3）如果 `FatherClassName1` 的所有父类没找到，就会搜索 `FatherClassName2`。
- （4）循环（2）、（3）两步，依次类推，直到找到为止。
- （5）如果都没找到就会报错。

Python 中采用了 C3 线性化算法去搜索父类，保证每个父类只搜寻一次，以避免搜索过程

陷入死循环。

9.4.2 实例 25：演示类的继承

下面通过实例来演示类的继承、子类方法覆写。

实例描述

创建一个父类，实现其初始化函数，再创建一个子类继承该父类。同时做如下实验。

(1) 验证子类继承父类的初始化函数（子类中没有实现初始化函数）：当通过子类实例化对象并传入初始值时，分析是否能实例化成功。

(2) 验证子类覆写父类方法：在子类与父类中实现同样函数名的方法，通过子类实例化对象，并调用该方法时，观察父类的方法是否被调用。

父类 Record 实现了__init__函数和 showrecode 方法。子类 GirlRecord 中实现了 showrecode 方法，没有实现__init__函数。在子类的 showrecode 方法中，调用了父类 showrecode 方法。代码如下：

代码 9-1：演示类的继承

| | |
|--|--------------------|
| class Record: | #定义一个类 |
| """A record class""" | #定义该类的说明字符串 |
| __Occupation = "scientist" | #职业为科学家，私有变量 |
| def __init__(self, name, age): | #定义该类的初始化函数 |
| self.name = name | #将传入的参数值赋值给成员变量 |
| self.age = age | |
| | |
| def showrecode(self): | #定义一个成员函数 |
| print("Occupation:",self.getOccupation()) | #该成员函数输出该类的成员变量 |
| | |
| def getOccupation(self): | #返回私有变量的方法 |
| return self.__Occupation | |
| | |
| class GirlRecord(Record): | #定义一个类子类 |
| """A GirlRecord class""" | #定义该类的说明字符串 |
| def showrecode(self): | #定义一个成员函数 |
| Record.showrecode(self) | #调用父类的方法 |
| print("the girl:",self.name,"'s age is",self.age) | #该成员函数输出该类的成员变量 |
| | |
| myc =GirlRecord ("Anna",42) | #对 GirlRecord 实例化 |
| myc.showrecode() | #调用其 showrecode 方法 |

对 `GirlRecord` 实例化，并调用 `showrecode` 方法。运行结果输出如下：

```
Occupation: scientist
the girl: Anna 's age is 42
```

第一行的输出是父类 `Record` 中的 `showrecode` 结果，第二行的输出是子类中 `showrecode` 方法的结果。

因为子类的 `showrecode` 中调用了父类的 `showrecode`，所以才会有父类的输出。

如果子类中不调用父类的方法，即将上面代码中倒数第 5 行（`Record.showrecode(self)`）注释掉，则将不会有父类方法 `showrecode` 的执行。这表明子类的 `showrecode` 方法已经对父类的 `showrecode` 方法进行了覆写。

9.4.3 super 函数

前面 9.4.2 小节中的例子，是一个单重继承，直接在子类的方法中调用父类即可调用父类函数。如果继承关系比较复杂，很容易出现父类方法被多次自动调用的情况。这是在编程过程中不希望发生的。为了避免这种情况出现，可以通过 `super` 函数来保证父类的方法只被执行一次。

`super` 函数本意是：获得父类的对象，并调用父类的方法。类似的，9.4.2 小节的例子代码中的倒数第 5 行，还可以写成以下样子：

```
super().showrecode()
```

使用 `super` 函数与直接使用父类（9.4.2 小节的例子代码中的倒数第 5 行的代码）的区别是：通过 `super` 函数进行调用的父类方法，会保证只执行一次。

在复杂继承关系中，`super` 函数会大大提高代码的可控性。下面通过正、反案例来说明。

9.4.4 实例 26：演示 super 函数的功能

下面通过实例来演示父类方法被多次调用的情况，以及如何使用 `super` 函数来避免父类的方法被多次调用。

实例描述

创建相对复杂的继承关系类结构：

- （1）创建一个父类，派生出两个子类，每个子类都需要调用父类的方法。
- （2）创建一个孙子类，继承这两个子类。孙子类也需要调用其每个父类的方法。

同时做如下实验：

- (1) 使用 9.4.2 小节的方法直接调用父类。通过输出来观察每个类的调用执行次数。
- (2) 使用 `super` 函数来实现子类对父类方法的调用，并通过输出来观察每个类的调用执行次数。

在这里，将通过两个程序片段来演示实例描述的两种情况。

1. 反面案例：直接调用父类，实现多重继承中的父类调用

使用 9.4.2 小节中的直接调用父类方法，实现在多重继承中对父类方法的调用。例子中的类结构如下：

- 父类为 `Record`。
- `Record` 派生的两个子类为 `FemaleRecord` 与 `RetireRecord`。
- `ThisRecord` 继承于 `FemaleRecord` 与 `RetireRecord`。

`Record`、`FemaleRecord`、`RetireRecord` 与 `ThisRecord` 这四个类都分别实现了各自的 `showrecode` 方法。具体如下：

- 孙子类 `ThisRecord` 的 `showrecode` 中，调用了其父类 `FemaleRecord` 与 `RetireRecord` 的 `showrecode` 方法。
- 子类 `FemaleRecord` 与 `RetireRecord` 的 `showrecode` 中，分别调用了其父类 `Record` 的 `showrecode` 方法。

具体见下面代码：

代码 9-2：演示 `super`

```

01 class Record:                                #定义一个父类
02     """A record class"""                    #定义该类的说明字符串
03     __Occupation = "scientist"                #职业为科学家，私有变量
04     def __init__(self, name, age):            #定义该类的初始化函数
05         self.name = name                     #将传入的参数值赋值给成员变量
06         self.age = age
07
08     def showrecode(self):                      #定义一个成员函数
09         print("Occupation:",self.getOccupation() )    #该成员函数返回该类的成员变量
10
11     def getOccupation(self):                  #返回私有变量的方法
12         return self.__Occupation

```

```

13
14 class FemaleRecord(Record):          #定义一个子类
15     """A GirlRecord class"""        #定义该类的说明字符串
16     def showrecode(self):            #定义一个成员函数
17         print(self.name,':',self.age," ,female" )      #该成员函数返回该类的成员变量
18         Record.showrecode(self)      #调用其父类的 showrecode 方法
19
20 class RetireRecord(Record):          #定义一个子类
21     """A RetireRecord class"""       #定义该类的说明字符串
22     def showrecode(self):            #定义一个成员函数
23         Record.showrecode(self)      #调用其父类的 showrecode 方法
24         print("retired worker" )     #该成员函数返回该类的成员变量
25
26 class ThisRecord(FemaleRecord,RetireRecord): #同时继承 FemaleRecord,RetireRecord
27     """A ThisRecord class"""         #定义该类的说明字符串
28     def showrecode(self):            #定义一个成员函数
29         print("the member detail as follow:" )          #该成员函数返回该类的成员变量
30         FemaleRecord.showrecode(self)                    #调用其父类的 showrecode 方法
31         RetireRecord.showrecode(self)
32
33 myc =ThisRecord ("Anna",62)
34 myc.showrecode()

```

程序的最后两行是对孙子类 `ThisRecord` 的实例化，并调用其实例化的 `showrecode` 函数。代码运行后，输出如下：

```

the member detail as follow:
Anna : 62 ,female
Occupation: scientist
Occupation: scientist
retired worker

```

可以看到，输出中的第 3、4 行是一样的内容。该内容是父类 `Record` 中 `showrecode` 函数输出的。这表明，父类 `Record` 中的 `showrecode` 函数被调用了两遍。

在实际编程过程中，这种父类函数被自动执行多次的情况是一定要避免的。如果 `showrecode` 函数中做了一些资源申请之类的操作，这种写法会导致资源泄漏，会严重地影响程序的性能。而且，代码中并没有调用两次的语句，这也大大增加了排查错误的困难。接下来，就使用 `super` 函数对该例子进行优化，避免这种情况的发生。

2. 正确案例：使用 super 函数，实现多重继承中的父类调用

编写代码，对 9.4.4 小节的“1. 反面案例”中的子类与孙子类内部的 showrecode 函数进行修改。将 showrecode 函数中对父类的引用部分都换成 super 函数。代码如下：

代码 9-2：演示 super（续）

```

35 class Record:                                #定义一个父类
36     """A record class"""                    #定义该类的说明字符串
37     __Occupation = "scientist"                #职业为科学家，私有变量
38     def __init__(self, name, age):            #定义该类的初始化函数
39         self.name = name                     #将传入的参数值赋值给成员变量
40         self.age = age
41
42     def showrecode(self):                      #定义一个成员函数
43         print("Occupation:",self.getOccupation() )    #该成员函数返回该类的成员变量
44
45     def getOccupation(self):                  #返回私有变量的方法
46         return self.__Occupation
47
48 class FemaleRecord(Record):                  #定义子一个类
49     """A GirlRecord class"""                #定义该类的说明字符串
50     def showrecode(self):                    #定义一个成员函数
51         print(self.name,':',self.age,"female" )    #该成员函数返回该类的成员变量
52         super().showrecode()
53
54 class RetireRecord(Record):                  #定义一个子类
55     """A RetireRecord class"""              #定义该类的说明字符串
56     def showrecode(self):                    #定义一个成员函数
57         super().showrecode()
58         print("retired worker" )            #该成员函数返回该类的成员变量
59
60
61
62 class ThisRecord(FemaleRecord,RetireRecord):    #定义一个类
63     """A ThisRecord class"""                #定义该类的说明字符串
64     def showrecode(self):                    #定义一个成员函数
65         print("the member detail as follow:" )    #该成员函数返回该类的成员变量
66         super().showrecode()
67
68 myc =ThisRecord ("Anna",62)
69 myc.showrecode()

```

再次运行代码，输出如下内容：

```
the member detail as follow:  
Anna : 62 ,female  
Occupation: scientist  
retired worker
```

可以看到，第三行的内容（“Occupation: scientist”）只输出了一次。表明父类 `Record` 中的 `showrecode` 函数被调用了一遍，程序达到了想要的效果。另外，在孙子类的函数 `showrecode` 中，使用 `super` 函数返回的是其继承的多个父类列表。系统会按照继承时的顺序，依次对每个父类进行 `showrecode` 方法的调用。

`super` 函数，是 Python 面向对象编程部分内置的一个非常有用的函数，它能提高程序的稳定性、可控性，也使得代码更为简洁。尤其在处理复杂继承关系的面向对象编程中，一定要将 `super` 函数应用起来。



注意：

使用 `super` 函数时，对父类的方法调用会自动传入 `self`。无需再传入 `self`，否则会报错误。

9.5 类相关的常用内置函数

Python 也提供了相关的内置函数来操作类。这里介绍几个常用的函数。

9.5.1 判断实例（`isinstance`）

函数 `isinstance` 的作用是，判断某个实例对象是否是某个类或其衍生类的实例。定义如下：

```
isinstance(object, class_name)
```

参数说明如下。

- `object`: 实例对象。
- `class_name`: 类名。

如果 `object`（`object.__class__`）是类 `class_name` 或其衍生类的实例，则返回 `True`。

该函数在前面 6.2.2 小节中也介绍过。

9.5.2 判断子类（`issubclass`）

函数 `issubclass` 的作用是，判断类是否是另一个类的子类。定义如下：

```
issubclass(class1, class2)
```

如果类 `class1` 是 `class2` 的派生类，则返回 `True`。例如 `issubclass(bool, int)` 会返回 `True`，因为 `bool` 是 `int` 的派生类。

9.5.3 判断类实例中是否含有某个属性（hasattr）

函数 `hasattr` 的作用是，判断类实例中是否含有某个属性。定义如下：

```
hasattr(obj, name, /)
```

如果类实例 `obj` 中含有 `name` 属性，则返回 `True`；否则返回 `False`。

9.5.4 获得类实例中的某个属性（getattr）

函数 `getattr` 的作用是，获得类实例中的某个属性。定义如下：

```
getattr(obj, name[, default])
```

如果类实例 `obj` 中含有 `name` 属性，则返回该属性的值；否则看是否有 `default`，如果有，则将该 `default` 的值返回；否则会产生一个 `AttributeError` 的异常。

该函数与 9.1 节中使用点符号直接访问类属性的功能类似，唯一不同的是多了一个默认值。

9.5.5 设置类实例中的某个属性值（setattr）

函数 `setattr` 的作用是，设置类实例中的某个属性。定义如下：

```
setattr(obj, name, value, /)
```

为实例 `obj` 中的 `name` 赋上 `value` 的值，如果 `obj` 中没有 `name`，则新建一个。该函数与 9.1 节中使用点符号直接访问类属性，并用等号为其赋值的功能几乎一样。其存在的价值在于，与 `getattr` 函数混合使用，如：

```
getattr(obj, name, setattr(obj, name, value, /))
```

该语句的作用是，先为 `obj` 中的 `name` 赋值 `value`（如果 `obj` 中没有 `name`，就创建一个，再赋值），接着再取 `obj` 中的 `name` 属性。

9.6 重载运算符

第4章介绍过 Python 中的各个基础类型，每个基础类型都有其特殊的操作方法（例如，列表类型的加、减操作与集合类型的加、减操作的意义就不一样）。

在 Python 内部，每个类型都相当于一个 Python 的类。在实现时，使用了一个叫作“重载运算符”的技术，来实现类型所对应的操作。

9.6.1 重载运算符的方法与演示

重载运算符，是指在类中定义并实现一个与运算符对应的处理方法。在两个对象进行运算符操作时，系统就会调用类中的具体方法来处理。下面通过代码来实现重载运算符：

```
class MyClass:                                #定义一个类
    """A record class"""                     #定义该类的说明字符串

    def __init__(self, name, age):             #定义该类的初始化函数
        self.name = name                     #将传入的参数值赋值给成员变量
        self.age = age

    def __str__(self):                         #用于将值转化为字符串形式，str(obj)
        return "name:"+self.name+";age:"+str(self.age)

    __repr__ = __str__                        #转化为供解释器读取的形式

    def __lt__(self,record):                   #重载 self< record 运算符
        if self.age < record.age:
            return True
        else:
            return False

    def __add__(self, record):                 #重载+号运算符
        return MyClass(self.name,self.age+record.age)

myc =MyClass ("Anna",42)                      #实例化一个对象 Anna，并为其初始化
myc1 = MyClass("Gary", 23)                   #实例化一个对象 Gary，并为其初始化

print(repr(myc))                             #格式化对象 myc，输出 "name:Anna;age:42"
print(myc)                                   #解释器读取对象 myc，调用 repr 输出 "name:Anna;age:42"
print(str(myc))                              #格式化对象 myc，输出 "name:Anna;age:42"
print(myc<myc1 )                            #比较 myc<myc1 的结果，输出: False
print(myc+myc1)                             #进行两个 MyClass 对象的相加运算，输出 "name:Anna;age:65"
```

在这个例子中，MyClass 类中重载了 repr、str、<、+运算符。用 MyClass 实例化了两个对象 myc 与 myc1。将 myc 进行 repr、str 运算，并打印其结果。从结果中可以看到，程序调用了重载的操作符方法__repr__、__str__。接着再让 myc 与 myc1 进行小于号、加号的运算，并打印其结果。从结果可以看到，程序调用了重载小于号的方法（__lt__），输出了 False。也调用了重载加号的方法（__add__），输出了“name:Anna;age:65”。



注意：

在前面 4.4.2 小节介绍过函数 repr，它可以将“转义字符”转化成“非转义字符”并输出。其实，那只是 repr 的一部分功能，repr 函数的参数可以是任何对象，不仅仅是字符串。

repr 函数的本意是，将输入的对象进行字符串化转换。当输入的是一个类时，就会将该类的类名及属性以字符串的形式显示出来。这部分内容可以参看“代码 9-4”。

9.6.2 可重载的运算符

表 9-1 列出了 Python 中支持可以重载的运算符，及其对应的方法名称。

表 9-1 可重载的运算符

| 重载的运算符方法名称 | 描 述 |
|----------------------------|---|
| __new__ | 创建类。在__init__之前创建对象 |
| __init__ | 构造函数。对象创建时，为其初始化 |
| __del__ | 析构函数。对象销毁时，进行收回操作 |
| __add__ | 加法运算符+。如果没有重载__iadd__，则在类似 X+Y、X+=Y 这样的语句中，加号生效 |
| __or__ | “或”运算符 。如果没有重载__ior__，则在类似 X Y、X =Y 这样的语句中，“或”符号的作用 |
| __repr__、__str__ | 格式转换。对应于函数 repr(X)、str(X) |
| __call__ | 函数调用。类似于 X(*args, **kwargs)语句 |
| __getattr__ | 点号运算。用来获取类属性 |
| __setattr__ | 属性赋值语句。类似于 X.any=value |
| __delattr__ | 删除属性。类似于 del X.any |
| __getattribute__ | 获取属性。类似于 X.any |
| __getitem__ | 索引运算。类似于 X[key]，X[i:j] |
| __setitem__ | 索引赋值语句。类似于 X[key]，X[i:j]=sequence |
| __delitem__ | 索引和分片删除 |
| __get__、__set__、__delete__ | 描述符属性。类似于 X.attr，X.attr=value，del X.attr |

续表

| 重载的运算符方法名称 | 描 述 |
|---|--|
| <code>__len__</code> | 计算长度。类似于 <code>len(X)</code> |
| <code>__lt__</code> , <code>__gt__</code> , <code>__le__</code> , <code>__ge__</code> , <code>__eq__</code> , <code>__ne__</code> | 比较。分别对应于 <code><</code> 、 <code>></code> 、 <code><=</code> 、 <code>>=</code> 、 <code>==</code> 、 <code>!=</code> |
| <code>__radd__</code> | 右侧加法。类似于 <code>other+X</code> |
| <code>__iadd__</code> | 加等于符号。类似于 <code>X+=Y</code> 中， <code>+=</code> 的作用 |
| <code>__iter__</code> , <code>__next__</code> | 迭代环境下，生成迭代器与取下一条。类似于 <code>I=iter(X)</code> 和 <code>next()</code> |
| <code>__contains__</code> | 成员关系测试。类似于 <code>item in X</code> |
| <code>__index__</code> | 整数值。类似于 <code>hex(X)</code> , <code>bin(X)</code> , <code>oct(X)</code> |
| <code>__enter__</code> , <code>__exit__</code> | 环境管理器，进入与退出。类似于 <code>with obj as var</code> 语法 |

9.7 包装与代理

本节介绍一个面向对象思想中相对高级的方法：对类的包装与代理。

9.7.1 包装

包装就是将某个对象（或类），进行重新打包，转换成另外一种更适合当前使用场合的对外接口。被包装后的对象（或类），自身的内部逻辑并没有改变，只是变化了对外的接口而已。这么做的初衷，是为了让对象（或类）更能适应当前的调用环境。

在 Python 中包装的具体做法是：

（1）创建一个类，将其要包装的目标类的对象，作为新创建类的成员对象。即，包装类包含被包装类的实例。

（2）为包装类实现各种与外部调用相吻合的接口。通过接口将调用关系传递到被包装类对象中，以实现真正的包装效果。

例如下列代码：

```
class WrapMe(object):
    def __init__(self, obj):      #初始化函数，将被包装对象传入
        self.__data = obj       #__data 为包装类的成员对象
    def get(self):                #返回被包装类中的数据，这里是返回整个对象了
        return self.__data
    def __str__(self):            #重载 str 函数
        #return str(self.__data)
        return "data="+str(self.__data)
```

```

mynum = WrapMe(888)           #对一个整型类的实例 888 进行包装，并实例化成 mynum 对象
print(str(mynum))             #调用了包装类对象的 str。输出：data=888

```

上面代码中，先通过初始化包装类 `WrapMe`，得到被包装类的对象 `mynum`（见倒数第 2 行）。

接着，调用包装类中所提供的 `__str__` 方法，来访问被包装类的私有化属性 `__data`。其中，`__str__` 方法，就是 `str` 函数的重载运算符处理函数。在 `__str__` 方法里，在返回的字符串前面加了一个 “data=”（见倒数第 4 行）。

于是得到了字符串 `data=888`，并输出。

9.7.2 代理

还可以使用代理的方式来实现类的包装。具体做法是，将包装类变成一个属性的“代理”。这使得访问包装类的属性等同于访问被包装类的属性。

下面通过代码演示代理的实现。在包装类中实现 `__getattr__` 函数，具体如下：

```

def __getattr__(self, attr):
    return getattr(self.__data, attr)

```

`__getattr__` 是 9.5.4 小节中 `getattr` 函数的重载函数（重载部分可以参看 9.6 节）。即，调用 `getattr` 函数时，会直接执行包装类中的 `__getattr__` 函数。例如以下代码：

```

class WrapMe(object):
    def __init__(self, obj):           #初始化函数，将被包装对象传入
        self.__data = obj             #__data 为包装类的成员对象
    def __repr__(self):                #重载 repr，转化为供解释器读取的形式
        return repr(self.__data)
    def __getattr__(self, attr):       #代理(delegation)
        return getattr(self.__data, attr)

Complex = WrapMe(2+4j)                #对一个复数类的实例进行包装，并实例化成 Complex 对象
print(Complex.real)                  #将实部输出：2.0

Complexlist = WrapMe([1,3])           #对一个列表类的实例进行包装，并实例化成 Complexlist 对象
Complexlist.append(4)                 #调用列表的属性方法 append
print(Complexlist)                   #输出结果：[1, 3, 4]

```

上面代码中，包装类 `WrapMe` 实现了 `__getattr__` 函数。

首先包装了一个复数，并实例化了对象 `Complex`。通过 `Complex` 代理可以直接访问复数的

real 属性，得到了复数 $2+4j$ 的实部，输出了 2.0。

接着又包装了一个列表，同样也是通过代理调用了列表的 `append` 方法，成功地添加了一个元素。这就是函数 `__getattr__` 的作用。通过 `__getattr__`，包装类就可以代理被包装的对象的所有属性了。

9.7.3 实例 27：使用代理的方式实现 RESTful API 接口

在使用 Python 开发 Web 项目时，通常情况下，需要为该 Web 程序开发对外可调用的 RESTful API 接口。

实现 RESTful API 接口的方法有多种，这里就来演示一下使用代理的开发方法。

1. 案例背景：RESTful API 介绍

在演示实例之前，有必要介绍下 RESTful API。RESTful API 是指在 RESTful 架构下的 API 接口。凡是遵循符合 REST 原则的软件架构，都被叫做 RESTful 架构。

表述性状态传递（Representational State Transfer，REST）是一种软件架构风格。它是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。它属于网站即软件的概念，即，在互联网环境中使用的单机版软件的思想来开发程序。

当前越来越多的 Web 服务开始采用 REST 风格设计和实现。每一个 RESTful API 都是一个 URI，代表一种资源。客户端和服务端之间，通过调用 RESTful API 来传递这种资源的某种表现层。客户端通过 GET、POST、PUT、DELETE 等动词实现的 HTTP 方法，对服务器端资源进行操作，实现“表述性状态传递”。

一般来讲，RESTful API 的形态如下：

```
http://example.com/customers/1234
http://example.com/orders/2007/10/776654
http://example.com/products/4554
http://example.com/processes/salary-increase-234
```

2. 案例介绍

从上面的 RESTful API 形态举例来看，RESTful API 其实是一个个的 URL 网址链接。假如给每个 URL 都写一个对应的方法，这样会是巨大的工作量；而且，一旦 API 需要改动，与其

对应的方法也得修改。这样的实现方式显然不合理。这种情况下，可以考虑使用代理的方式，对 RESTful API 接口进行动态地生成，以简化开发的工作量。

实例描述

使用代理的方式实现 RESTful API 的两种形式接口：静态接口与动态接口。静态接口指的是固定的 URL；动态接口指的是带有参数的 URL，类似 GitHub 的 API 会把参数放到 URL 的中间。

(1) 实现静态接口：/status/allusers/list 接口。

(2) 实现动态接口：/users/:user/info。其中，冒号后面的 user 指代具体的参数。

这里将通过两个程序片段来演示“实例描述”中的两种情况。

3. 实现静态接口

类似于 9.7.2 小节中的代理代码，创建一个类 ChainURL，重载 __init__ 与 __getattr__ 运算符。并在 __getattr__ 内部返回字符串拼接好后的 ChainURL 类。这样，令 ChainURL 为自己的属性代理，就实现了拼接字符串的功能。代码如下：

代码 9-3：演示 RESTful API 接口

```
01 class ChainURL(object):
02     def __init__(self, attr=''):          #初始化函数，传入被包装对象
03         self._data = attr                #data 为包装类的成员对象
04
05     def __getattr__(self, attr):          #定义代理函数(delegation)
06         return ChainURL('%s/%s' % (self._data, attr)) #返回自身对象，实现链式字符串拼接
07
08     def __str__(self):
09         return self._data
10
11     __repr__ = __str__
12
13 print(ChainURL().status.allusers.list) #将拼接好的接口打印出来
```

上面的代码运行后显示如下：

```
/status/allusers/list
```

上面的代码 9-3 中，通过对类 ChainURL 重载 __getattr__ 运算符，并返回一个使用拼接好的字符串进行初始化的 ChainURL 类，以实现 URL 链的生成。

最后一行代码的执行过程如下：

- (1) 程序先执行 `ChainURL()`，返回一个内部属性 `_data` 为空的 `ChainURL` 类型。
- (2) 执行 `ChainURL` 类型的 `status` 属性访问，程序自动进入 `__getattr__` 中，并返回一个内部属性 `_data` 为 `“/status”` 的 `ChainURL` 类型。
- (3) 重复上面步骤，返回一个内部属性 `_data` 为 `“/status/allusers”` 的 `ChainURL` 类型。
- (4) 重复上面步骤，返回一个内部属性 `_data` 为 `“/status/allusers/list”` 的 `ChainURL` 类型。
- (5) 在 `print` 函数中，该 `ChainURL` 类型变量调用了内部 `__repr__` 函数。
- (6) `__repr__` 与 `__str__` 一致，于是转嫁调用 `__str__` 函数，最终返回了内部 `_data` 的值，并打印出来。

4. 实现动态接口

要想实现类似 `/users:/user/info` 的 URL，首先可以将其转化为对应的链式表达式代码。

这里提供的一个思路是，将其中的参数作为一个函数调用的参数传入。这样对应的链式表达式可以写成 `ChainURL().users(user).info`。要想让这行代码工作，需要再重载 `__call__` 运算符，使 `ChainURL` 类具有函数的属性，即可以被调用。这里将传入的参数 `user` 实例化为 `“Anna”`。代码如下：

代码 9-3：演示 RESTful API 接口（续）

```

14 class ChainURL(object):
15     def __init__(self, attr=''):          #初始化函数，将被包装对象传入
16         self._data = attr                #data 为包装类的成员对象
17
18     def __getattr__(self, attr):          #定义代理函数(delegation)
19         return ChainURL('%s/%s' % (self._data, attr)) #返回自身对象，实现链式字符串拼接
20
21     def __call__(self, attr):             #重载 call 运算符，使其具有被调用的属性
22         return ChainURL('%s/%s' % (self._data, attr))
23
24     def __str__(self):
25         return self._data
26
27     __repr__ = __str__
28
29 print(ChainURL().users('Anna').info)    #将拼接好的接口打印出来

```

上面的代码运行后显示如下：

```
/users/Anna/info
```

上面的代码 9-3（续）中，通过对类 `ChainURL` 重载 `__call__` 运算符，并返回一个使用拼接好的字符串进行初始化的 `ChainURL` 类，以实现 URL 链中对参数“Anna”的接收。

在最后一行代码的执行过程如下：

（1）程序先执行 `ChainURL()`，返回一个内部属性 `_data` 为空的 `ChainURL` 类型。

（2）执行 `ChainURL` 类型的 `status` 属性访问，程序自动进入 `__getattr__` 中，并返回一个内部属性 `_data` 为“/users”的 `ChainURL` 类型。

（3）代码会解析到 `ChainURL` 类型后面的括号，认为是一个函数调用操作。于是将括号内的“Anna”传入重载的操作符 `__call__` 中，返回一个内部属性 `_data` 为“/users/Anna”的 `ChainURL` 类型。

（4）进入 `__getattr__` 中，返回一个内部属性 `_data` 为“/users/Anna/info”的 `ChainURL` 类型。

（5）使用 `print` 函数将拼接好的 URL 打印出来。

9.8 自定义异常类

这部分内容需要用到前面第7章的知识。在 Python 中，异常也是以类的形式存在的。在特殊情况下，Python 中自带的一些异常类无法满足编程的需要。这时，可以自定义异常类。

下面就来介绍自定义异常类的方法。

9.8.1 自定义异常类的方法

Python 中的异常类都是从 `Exception` 类派生的。自定义的异常类也不例外。当然，自定义的异常类也可以从 `Exception` 类间接继承，即从 `Exception` 的子类派生。

自定义的异常类方法如下：

```
class MyEpt(Exception):           #定义一个自定义的异常类 MyEpt
    def __init__(self, value):    #重载初始化运算符
        self.value = value
    def __str__(self):           #重载 str
        return repr(self.value)
```


上面的代码自定义了一个异常类。与一般类的区别在于，`MyEpt` 后面加了个父类 `Exception`。这表明 `MyEpt` 是从父类 `Exception` 继承的。这样 `MyEpt` 就有了异常类的属性。其中，重载运算符 `__init__` 是为了让该异常还可以接收输入参数；而重载运算符 `__str__` 的作用是将该参数的内容输出，可以用于屏幕输出，或是日志，或是对接后面的其他程序。

自定义异常类的使用方法，与第 7 章介绍的异常方法完全一致。例如：

```
try:                                #使用 try 语法，开始准备捕获异常
    raise MyEpt('some thing is wrong') #使用 raise 语法来创建一个异常
except MyEpt as e:                  #捕获到 MyEpt 异常
    print('My exception:', str(e)) #打印该异常，输出"My exception: some thing is wrong"
```

上面代码，使用了 `raise` 来创建一个自定义类型的异常，同时又使用 `except` 将其捕获，并输出传入的异常内容。

9.8.2 实例 28：自定义异常类的多重继承与使用

在实际应用中，常会在一个模块里，创建一个自定义异常类作为基类，然后派生出各种不同的子异常类。这种写法可以使程序层次分明、更加结构化。下面通过实例演示。

实例描述

定义一个异常基类 `MyEpt`，并实现其派生类 `MyEptInput` 与 `MyEptQueue`。进行如下操作：

(1) 分别创建三个类的异常——`MyEpt`、`MyEptInput`、`MyEptQueue`，并依次捕获，观察其输出结果。

(2) 在创建异常时，分别传入整型与字符型参数，并运行观察其输出结果。思考结果产生的原因。

具体做法及代码如下。

1. 自定义异常类多重继承的代码实现

基类实现对 `Exception` 的继承，不做任何其他的操作。派生类 `MyEptInput` 继承了 `MyEpt`，并重载了初始化函数；派生类 `MyEptQueue` 继承了 `MyEpt`，并重载了初始化与字符串转换函数。代码如下：

代码 9-4：自定义异常类的多重继承与使用

```
class MyEpt(Exception):
    """Base class for exceptions in this module."""
```

```

pass

class MyEptInput(MyEpt):
    def __init__(self, value):
        self.value = value
#定义一个自定义的异常类 MyEptInput
#重载初始化运算符

class MyEptQueue(MyEpt):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return str(self.value)
#定义一个自定义的异常类 MyEptQueue
#重载初始化运算符
#重载 str

for e in [MyEpt, MyEptInput, MyEptQueue]:
    try:
        raise e(111)
    except MyEptQueue as e:
        print("MyEptQueue:", str(e))
    except MyEptInput as e:
        print(repr(e))
    except MyEpt as e:
        print(e)
#使用 for 循环来依次生成不同类型的异常
#通过 try、except 语句进行异常抛出和捕获
#生成异常，并传入整型参数
#对 MyEptQueue 类型异常捕获，输出其字符串化后的内容
#对 MyEptInput 类型异常捕获，输出其字符串化后的内容
#对 MyEpt 类型异常捕获，输出其内容

```

上面代码中，使用 for 循环来生成不同类型的异常。异常的顺序依次为 MyEpt、MyEptInput、MyEptQueue，并将其捕获。根据不同类型显示不同的内容。代码运行后，生成结果如下：

```

111
MyEptInput(111,)
MyEptQueue: 111

```

对于输出结果的解读如下：

- 第 1 行显示为“111”。这表明是异常类 MyEpt 内容的生成结果，其中，在 print 内部调用了父类 Exception 的 repr 重载函数。
- 第 2 行显示为“MyEptInput(111,)”。这表明是异常类 MyEptInput 的内容输出，是函数 repr 将 MyEptInput 的名字与属性字符串转化后的结果。
- 最后 1 行显示为“MyEptQueue: 111”。这表明是异常类 MyEptQueue 的内容输出，在输出过程中调用了重载函数 str。

2. 异常类对于字符串参数的处理

将上面的代码中倒数第 7 行的 raise e(111) 改成 raise e('111')。运行代码，可以看到如下结果：

```
111
MyEptInput('111',)
MyEptQueue: 111
```

输出结果的第 2 行中，字符串 111 的两边带有单引号，而第一行和最后一行的 111 都不带单引号。这表明，异常类 `Exception` 中的 `repr` 方法不会对传入的参数进行改变。而 Python 中内置的 `repr` 函数，会将类实例内部 `str` 类型的属性上加一个单引号，使其变为字符串。内置函数 `repr` 的做法更为严谨，因为它可以将类实例对象中属性的类型区分开来。不过，在使用内置函数 `repr` 来转化结果时，还是要考虑引号的存在。

3. 附加实验

想一想，如果将 `MyEpt` 类放在第一个接收异常的地方，生成的结果会是怎样？

即，前面“1. 自定义异常类多重继承的代码实现”中最后 6 行代码改成如下样子：

```
except MyEpt as e:
    print(e)                                #对 MyEpt 类型异常捕获，输出其内容
except MyEptQueue as e:
    print("MyEptQueue:",str(e))             #对 MyEptQueue 类型异常捕获，输出其字符串化后的内容
except MyEptInput as e:
    print(repr(e))                          #对 MyEptInput 类型异常捕获，输出其字符串化后的内容
```

这种情况下，代码将永远不会进入 `MyEptQueue` 与 `MyEptInput` 中。程序运行后的输出结果会是：

```
111
111
111
```

这是因为，在异常捕获过程中，子类的异常也会去匹配父类。而 `except` 语句是按照从上往下的顺序来匹配的，于是所有的异常就都进入了 `MyEpt` 分支。全部输出 111。

这种情况会导致派生类异常全部失效。这是在编写程序中一定要避免的情况。

9.9 支持 with 语法的自定义类

在前面 8.4 节中介绍过，`with` 语法可以使代码变得更简洁，使得开发者只需要关心事先、事中的事情，可以不用关心事后事情。例如，`with` 语句可以让文件对象在使用后被正常地关闭。

9.9.1 实现支持 with 语法的类

在 8.4.2 小节介绍过，with 语句并不适用于所有对象，能够使用 with 语句的对象必须有一个__enter__方法、一个__exit__方法。

对于自定义类来讲，要想实现这个功能，需重载这两个方法：在__enter__里实现初始的操作（相当于文件对象中的打开文件句柄操作）；在__exit__里实现结尾的清理工作（相当于文件对象中的关闭文件句柄操作）。具体的实现请参考下面的实例。

9.9.2 实例 29：代码实现自定义类，并使其支持 with 语法

在实际情况中，__exit__方法不仅要去做结尾的清理工作；如果在 with 语句进行的过程中，出现异常而被迫退出时，也是会调用__exit__方法。而这时的__exit__方法就将异常值透传出去，并在屏幕上输出。

下面通过实例来演示自定义类的代码实现，并令其支持 with 语法。

实例描述

定义一个类 withClass，并重载__enter__与__exit__方法。进行如下操作：

（1）在__enter__与__exit__方法中输出打印信息，使用 with 语句运行程序，观察其输出结果。

（2）在__exit__方法中输出异常值的相关信息，在 with 语句内部制造异常并运行，观察其输出结果。

编写类的代码。先进行正常的 with 语句调用，具体代码如下。

1. 自定义类支持 with 语法的代码实现

在类 withClass 中实现了三个方法：

- 重载__enter__方法：打印相关信息，并返回实例化对象；
- 重载__exit__方法：打印相关信息，同时将参数 type、value、trace 一起打印出来（这三个参数为产生异常时的异常信息）；
- 自定义的成员方法 work：为了演示方便，供 withClass 的实例化对象在 with 语句块中调用。

代码如下：

代码 9-5：实现支持 with 语法的类

```

class withClass:                                #自定义类
    def __enter__(self):                        #重载 enter__
        print ("__enter__")
        return self                            #将实例化对象返回

    def __exit__(self, type,value, trace):      #重载__exit__
        print ("__exit__")
        print ("type:", type)
        print ("value:",value)
        print ("trace:",trace)

    def work(self):                             #自定义函数，输出一句话
        print('Hello,Python!')

with withClass() as myclass:                    #使用 with 语法将函数的返回值赋给 myclass
    myclass.work()                             #实现对 withClass 成员函数的调用

```

上面代码执行后，输出如下结果：

```

__enter__
Hello,Python!
__exit__
type: None
value: None
trace: None

```

在代码 9-5 倒数第 2 行的 with 语句中：

(1) withClass()语句执行时，会打印出__enter__信息，并返回实例化对象。

(2) as 语句将返回的实例化对象赋值给 myclass。

(3) 代码执行最后一句话，输出“Hello, Python”。

(4) 准备退出 with 语句之前，会调用 withClass 中的__exit__方法，打印出__exit__信息。

因为 with 代码块中的程序运行正常，所以在__exit__方法中输出的三个参数 type、value、trace 的值都为空。

2. with 块中异常情况的实例演示

代码 9-5 中的 work 方法里（倒数第 4 行）加一句代码，调用 raise 函数抛出一个异常。如下：

```
def work(self):                                #自定义函数，输出一句话
    print('Hello,Python!')
    raise(ValueError('value wrong!'))         #抛出一个异常
```

其他地方的代码均不用变。整体运行后，输出如下内容：

```
__enter__
Hello,Python!
__exit__
type: <class 'ValueError'>
value: value wrong!
trace: <traceback object at 0x000000000D74EC08>
Traceback (most recent call last):
.....
```

在 with 后面的代码块抛出任何异常时，__exit__方法被执行。并将与之关联的三个参数 type、value、trace 一起传给__exit__方法，于是在__exit__方法中可以将相应的值打印出来。

9.10 “自定义迭代器类”的实现与调试技巧

在 5.8 节中，介绍了 for 的原理——迭代器，在了解了迭代器的机制后，结合本章内容，就可以实现一个自己的迭代器类了。

9.10.1 实例 30：自定义迭代器，实现字符串逆序

实例描述

定义一个类，将输入的字符串逆序输出，同时支持迭代器功能。使用普通字符串对该类进行实例化，并做如下操作：

- (1) 使用 for 对迭代器对象进行迭代，依次将迭代的内容输出，观察其输出结果。
 - (2) 在 for 循环退出后，再次调用__next__函数，观察其输出结果。
-

具体代码如下。

1. 迭代器的实现及使用

在迭代器类 Reverse 中，重载初始化函数__init__，并为自身私有成员变量赋值。重载迭代器函数__iter__与获取元素函数__next__。在__next__中，使用对私有变量__index 每次减一的方法，实现从后往前返回字符。并且在__index 为 0 时，抛出 StopIteration 异常。代码如下：

代码 9-6：自定义迭代器类演示

```

01 class Reverse:                                #自定义迭代器
02     """ 逆序迭代一个序列 """
03     def __init__(self, datastr):                #重载初始化函数
04         self.__data = datastr                  #定义私有变量data 与 index
05         self.__index = len(datastr)
06     def __iter__(self):                        #重载迭代器函数
07         return self
08     def __next__(self):                        #重载获取下一元素函数
09         if self.__index == 0:
10             raise( StopIteration )            #当元素取完之后，再次获取元素，抛出异常
11         self.__index -= 1
12         return self.__data[self.__index]
13
14 revstr = Reverse('Python')                    #创建类实例对象
15 for c in revstr:                              #使用 for 循环，遍历迭代器
16     print(c, end=' ')                        #输出: n o h t y P

```

上面代码运行后，输出如下内容：

```
n o h t y P
```

代码 9-6 中，自定义类 `Reverse` 按照实例化字符串“Python”的值对自身初始化（倒数第 3 行）。并通过重载迭代器函数 `__iter__`（代码第 6 行）与获取元素函数 `__next__`（代码第 8 行），完成对 `for` 循环的支持。在 `__next__` 中，每次返回最后一个元素，并将原始索引由最后逐渐向前移动来进行字符串的逆序，得到最终的输出结果。

2. 迭代终止异常的触发

在代码 9-6 的基础上，再添加一句调用 `__next__` 的代码，如下：

```
print(revstr.__next__())
```

代码运行后，程序出现了异常，并且终止。这是因为，当 `for` 语句跳出之后，实例 `revstr` 的成员变量 `__index` 已经为 0。再次运行时，`__index` 并没有改变，于是仍然满足等于 0 的条件，同样抛出异常。



注意：

编写迭代器最容易忽视的一个环节是：在自定义类中加入对循环结束的判断，并抛出 `StopIteration` 异常（代码 9-6 倒数第 7 行）。只有这么做了，`for` 循环语句才会接收到 `StopIteration` 异常，并当作终止信号来结束循环。

9.10.2 调试技巧

在 5.8 节中，介绍过 for 循环的原理。这里借助“代码 9-6：自定义迭代器类演示”，进行 for 调用过程的调试，从而来介绍一下 Python 调试方面的知识。

以 Anaconda 的图形化界面操作为例。该调试过程分为如下 4 步：

(1) 添加断点：属于程序以外的操作，不会影响代码，但是程序在调试下运行时，会在断点处暂停。

(2) 启用调试模式运行：使程序以调试的方式运行。

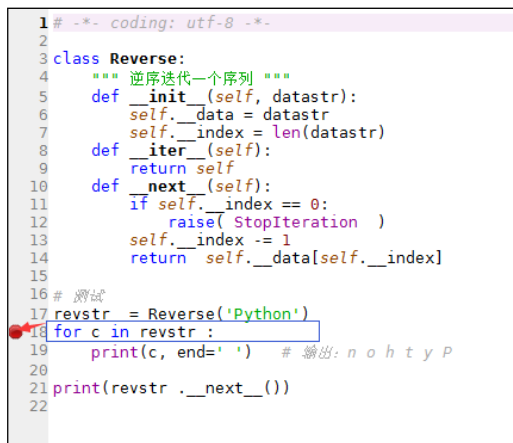
(3) 查看内存值：在程序暂停后，查看当前相关变量的值。通过人为的计算与分析来判断程序运行的逻辑是否正常。

(4) 单步执行：让程序接着再往下执行一步，如果该步骤是函数调用，还会有 3 种可选操作——进入函数里面执行、把整个函数当作一个语句来执行、跳出当前函数进入到上层代码块来执行。

目的是通过该部分的操作，来学会在程序运行状态下，查看内部调用顺序、数值变化的方法，掌握解决 BUG 的能力。

1. 添加断点

找到要调试的具体代码（for 语句），并为该行添加断点，如图 9-1 所示。



```

1 # -*- coding: utf-8 -*-
2
3 class Reverse:
4     """ 逆序迭代一个序列 """
5     def __init__(self, datastr):
6         self.__data = datastr
7         self.__index = len(datastr)
8     def __iter__(self):
9         return self
10    def __next__(self):
11        if self.__index == 0:
12            raise( StopIteration )
13        self.__index -= 1
14        return self.__data[self.__index]
15
16 # 测试
17 revstr = Reverse('Python')
18 for c in revstr :
19     print(c, end=' ') # 输出: n o h t y P
20
21 print(revstr.__next__())
22
  
```

图 9-1 添加断点

在图 9-1 中第 18 行的位置的左边区域双击鼠标，则在该鼠标的双击位置出现一个红色的圆点。这个圆点就叫作断点。

2. 启用调试模式运行

添加断点后，进入调试模式运行，如图 9-2 所示。单击图中 9-2 箭头所指的按钮，即可打开调试模式运行。

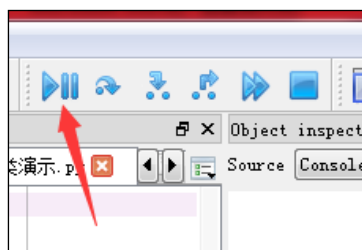


图 9-2 单击按钮

单击图 9-2 中的按钮之后，程序会在设置了断点的地方停下来。具体界面如图 9-3 所示。

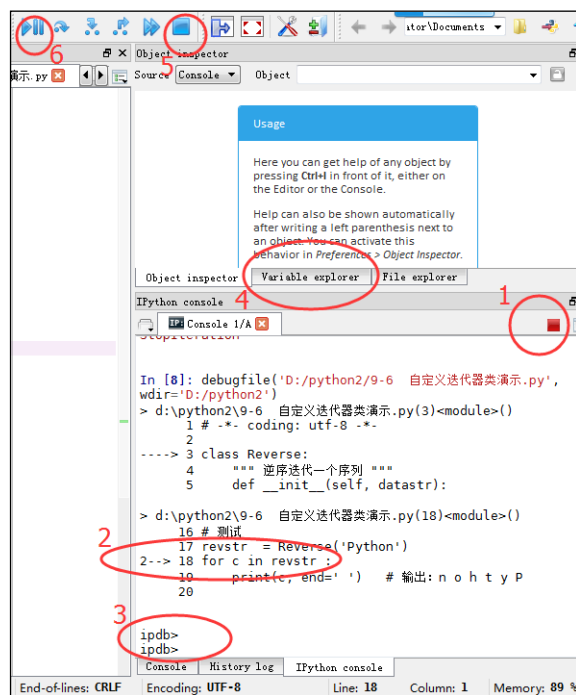


图 9-3 在断点停下

图 9-3 中，标注“1”的按钮是红色，表明当前程序是运行状态。再次单击它，就会变成灰色，表明当前程序没有在运行。同样也可以单击标注为“5”的按钮来停止程序运行。

在当前情况下，如果在单击“启动”按钮（图 9-3 中标注为“5”的按钮），程序就会接着该断点的地方继续往下运行。如果是程序在运行过程中单击“启动”按钮，那便是暂停命令，程序会暂停到当前运行的代码位置。

图 9-3 中标注“2”的部分，是向用户交代当前程序暂停的位置。

图 9-3 中标注“3”的部分是 ipdb 调试命令。可以通过输入调试命令后按 Enter 键的方式，进行各种调试操作，与界面操作的功能相同。

3. 查看内存值

单击图 9-3 中标注“4”的按钮，即可看到当前内存中变量的值，如图 9-4 所示。

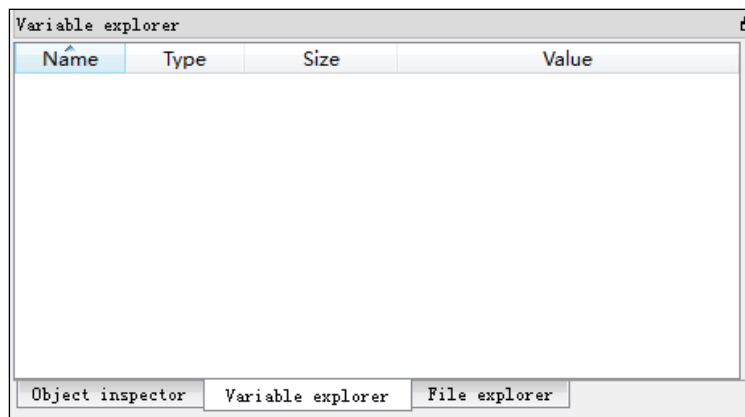


图 9-4 看到变量的值

图 9-4 中并没有任何内容，原因是当前该句代码还没有执行，并没有创建变量。这时需要让代码单步执行下一条语句。

4. 单步执行

为了更清楚地了解 for 内部的调用过程，单击图 9-5 中的标注“2”的按钮。



图 9-5 单击按钮

图 9-5 中标注的三种按钮就是单步执行的三种情况：“1”代表把整个函数当作个语句来执行；“2”代表进入函数里面执行；“3”代表跳出当前函数，进入到上层代码块来执行。

在单击图 9-5 中标注“2”的按钮后，可以看到断点的位置发生了变化，如图 9-6 所示。

图 9-6 中箭头的内容为函数__iter__，这表明 for 中的第一步进行了对__iter__的调用。接着继续单击 3 次图 9-5 中的标注“1”的按钮，程序来到了__next__调用的位置，如图 9-7 所示。

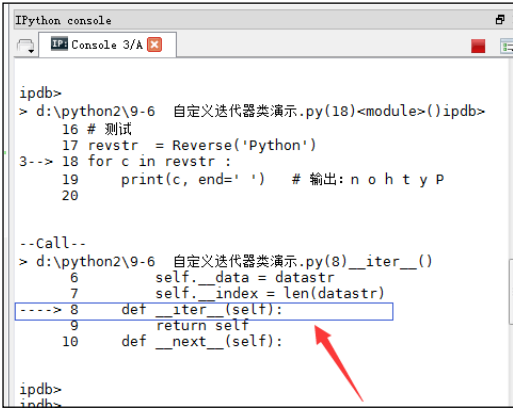


图 9-6 断点位置发生变化

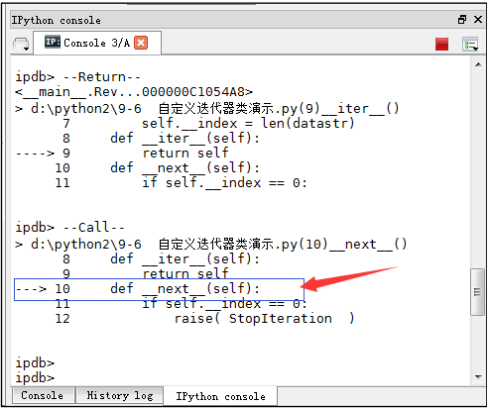


图 9-7 调试到__next__调用的位置

这表明了：for 语句中，调用__iter__得到迭代器之后，又去调用该迭代器的__next__函数，开始获取数据。用鼠标连续单击 5 次图 9-5 中标注“1”的按钮之后，程序来到了原始 for 循环的下一句，如图 9-8 所示。

这时在变量窗口可以看到出现了变量 c 的信息，类型为 str，值为 n。

以上就是调试程序的必要步骤。接下来再介绍一下 ipdb 命令行方式所支持的具体命令。

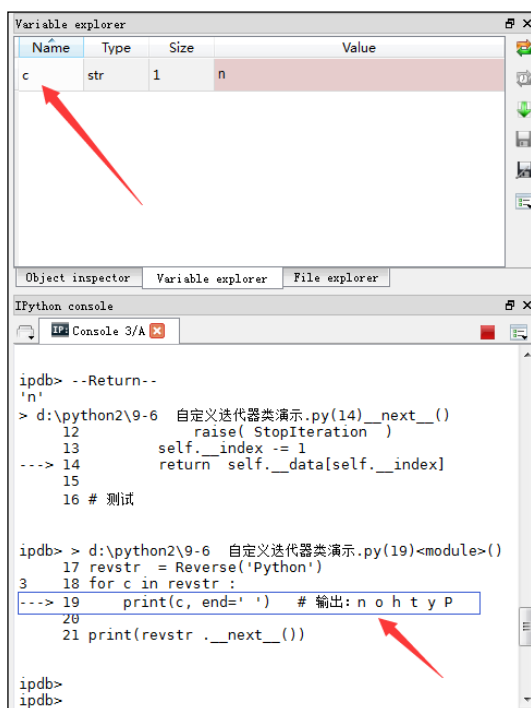


图 9-8 到达原始 for 循环的下一句

5. 用命令行方式调试 Python 程序 (ipdb 命令)

Anaconda 也支持以命令行的方式进行调试。在图 9-3 中标注“3”的位置输入 `p` 命令，后面跟上要查看的变量名 `c`，即可将 `c` 的值输出。例如：

```
ipdb> p c
'n'
```

同样，还可以用“`!Python 语句`”的方式，在当前环境下执行代码。例如，下面的命令就是执行一条语句，语句内容为：将变量 `c` 用 `print` 函数打印出来。

```
ipdb> !print(c)
n
```

相比较界面交互的调试方式，命令行方式的应用场景更加广泛。它适用于习惯在 `linux` 下操作或是通过远程 `shell` 命令进行操作的用户。其实，界面的交互操作，最终也是调用了 `ipdb` 命令。可以说，命令行方式是界面交互的底层实现。更多 `ipdb` 命令见表 9-1。

表 9-1 ipdb 命令

| ipdb 命令 | 描 述 |
|---------|--|
| enter | (按 Enter 键) 重复上次命令 |
| c | 继续运行 |
| l | 查找当前位于哪里 |
| s | 进入子程序 |
| r | (run) 运行直到子程序结束 |
| ! | 执行 Python 命令 |
| h | (help) 帮助 |
| a | (args) 打印当前函数的参数 |
| j | (jump) 让程序跳转到指定的行数 |
| l | (list) 列出当前将要运行的代码块 |
| n | (next) 让程序运行下一行，如果当前语句有一个函数调用，用 n 是不会进入被调用的函数体中的 |
| p | (print) 最有用的命令之一，打印某个变量 |
| q | (quit) 退出调试 |
| r | (return) 继续执行，直到函数体返回 |
| s | (step) 跟 n 相似。但如果当前有一个函数调用，则 s 会进入被调用的函数体中 |

9.11 元类（MetaClass）

元类属于 Python 语法中的高级内容。如掌握了元类的相关知识，则对 Python 中类的实现机制有更深刻地了解。

元类方面的内容，在设计一些底层库时常常会用到，因为它可以使设计变得更简洁。当然，如果读者只需使用 Python 完成一些应用层逻辑，可以跳过本节。

9.11.1 Class 的实现原理

在 4.3.1 小节中介绍过，函数 type 的作用是获取对象类型。Python 中类 class 的实现，也是通过 type 函数来完成的。使用 help 函数查看 type，会得到如下信息：

```
01 help(type)
02 Help on class type in module builtins:
03 class type(object)
04 |   type(object_or_name, bases, dict)
05 |   type(object) -> the object's type
06 |   type(name, bases, dict) -> a new type
07 .....
```

第3行开始就是 `type` 的定义。在 Python 中，`type` 有3种形式（见上面代码的第4、5、6行）。其中，第5行显示的 `type` 函数，就是前面 4.3.1 小节中返回对象的类型；而第4、6行的 `type` 函数，是动态创建一个 `class`。以第6行为例，共有3个参数。

- **name**: 类的名称。
- **bases**: 类的父类（可以是多个）。
- **dict**: 类的成员函数（可以是多个）。

下面通过一段代码进行演示：

```
def fun(self):                # 定义一个函数，输出 Hello Python
    print("Hello Python!")

cls = type('MyClass', (object,), dict(myfun=fun)) #使用 type 创建类 cls
obj = cls()                    #进行类 cls 的实例化
obj.myfun()                   # 调用类的成员函数 myfun，转嫁调用 fun，最终输出: Hello Python!
```

上面的代码中，倒数第三行使用了 `type` 函数创建了一个类，并用变量 `cls` 来作为类的引用。在这行代码中，向 `type` 函数传入了3个参数，代表类 `cls` 的组成：类名是 `MyClass`，父类是 `object`，成员函数是 `myfun`。

其中，在成员函数（`myfun`）的表达上，执行了赋值语句 `myfun=fun`。该语句的作用是将函数 `fun` 绑定到 `myfun` 上。这样，通过对类实例对自己内部成员函数 `myfun` 进行调用时，系统会自动去执行函数 `fun` 的代码（见最后一行）。输出了 `Hello Python!` 的内容。

系统运行到关键字 `class`，内部会去调用 `type` 函数来生成类。所以，使用 `type` 函数生成的类与使用 `class` 生成的类完全一样，可以通过打印其属性值来查看具体的内容。例如：

```
print(cls.__name__)          # MyClass
print(cls.__bases__)         # (<class 'object'>,)
print(cls.__dict__)          # {......, 'myfun': <function fun at 0x00000000D788950>,.....}
```

9.11.2 元类的介绍

函数 `type`，适合于动态地创建相对简单的类。

若要创建更复杂的类，则需要通过元类（`MetaClass`）的方式。元类，就是创建类的类。即，在创建类之后，再由类来创建实例进行应用。

1. 元类的定义

定义元类时，需要继承于 `type` 类；默认的命名习惯是，让类名以 `MetaClass` 结尾；在元类中需要定义并实现 `__new__` 方法（一定要有返回值）。使用元类来创建类时，元类中的 `__new__` 方法将会被调用，用于生成新建的类。`__new__` 方法中有以下几个必要的参数。

- `cls`: 类的对象。
- `name`: 类的名字。
- `bases`: 类继承的父类集合。
- `attrs`: 类的属性。

`__new__` 方法中，可以通过对 `attrs` 的修改，来实现类属性的控制。具体的实现如下：

```
class MyClassMetaClass (type):                                # 定义元类
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: print("add value %s" % value)    #添加
add 属性,
        attrs['name'] = 'personName'                                     #给类添加 name 属性
        return type.__new__(cls, name, bases, attrs)                # 通过 type 模块来创建新的类
```

上面代码定义了元类 `MyClassMetaClass`，并添加了两个属性 `add` 与 `name`。`add` 为一个函数，`name` 为一个字符串。

之所以要继承 `type` 并实现 `__new__` 方法，是因为在创建类时，内部调用了 `type` 的 `__new__` 方法为这个类分配内存空间。当内存分配好后，便会调用 `type` 的 `__init__` 方法初始化（做一些赋值等）。

2. 元类的使用

使用元类创建新类时，需要在新类名后面的括号里传入元类名称，并将传入的元类名称赋值给形参 `metaclass`。接上面“1. 元类的定义”中的例子，添加如下代码：

```
class MyClass(object, metaclass=MyClassMetaClass):# 通过 metaclass 指定该类对应的元类
    pass
```

在代码中，使用元类 `MyClassMetaClass` 动态创建了一个新的类 `MyClass`。之后就可以正常使用新建类 `MyClass` 来进行实例化对象了。下面是访问 `MyClass` 属性的代码：

```
obj = MyClass()                #实例化对象
obj.add(55)                    #输出: add value 55
print(obj.name)               #输出: personName
```

上面的代码中，第1行是将 `MyClass` 实例化为 `obj`。第2、3行是访问实例化对象 `obj` 的 `add` 与 `name` 属性。



注意：

在 Python 的早期版本中，元类的使用方法并不是这样的，是直接在类中设定一个 `__metaclass__` 属性，让它等于 `MyClassMetaclass`，即 `__metaclass__ = MyClassMetaclass`。这种写法在 Python 3.5 后已经不再生效。读者在自学本书的同时，如果遇到低版本 Python 实现的元类代码，记得将此处修改，方能正常运行。

3. 元类与继承的区别

元类的写法与继承很像，作用也很像。但是从意义来讲，两者有很大的差别，见下面代码：

```
print(isinstance(MyClass, MyClassMetaclass))    #判断是否是实例化关系，输出：True
print(issubclass(MyClass, MyClassMetaclass))    #判断是否是派生关系，输出：False
```

可以看到，元类与被创建的新类是实例化关系，并不是派生关系。



注意：

如果把上面的 `MyClassMetaclass` 换成 `object`，会发现第一句 `print(isinstance(MyClass, object))` 的结果仍然是 `True`。这和继承关系的规则相违背。这是个特例，因为 Python 中所有的对象和类都是 `object` 的实例，所以该行代码返回值是 `True`。除 `object` 外，其他的继承关系的两个类，使用 `instance` 函数进行判断时，返回都是 `False`。

4. 元类的意义

可以思考一下：在 9.11.2 小节的例子代码中，假如直接在 `MyClass` 上添加两个属性，也可以实现同样的效果。那么元类的意义是什么？

其实上述例子可以理解成，元类 `MyClassMetaclass` 对 `MyClass` 类进行了属性的添加。这便是元类最主要的作用——它可以更方便地对类属性进行修改。

利用这一特性，可以方便地编写出更为高效而合理的底层框架代码。例如：“对象-关系”映射（Object Relational Mapping, ORM）就是一个典型的例子。ORM 是常用来读/写数据库方面的技术。即，一个类对应于数据库中的一个表。通过这样的写法，来实现持久层与控制层的分离。这样，程序员在编写应用层代码时，可以专心考虑应用层的业务逻辑，无需通过 SQL 语句操作数据库。

第 10 章

系统调度——实现高并发的处理任务

无论是操作系统，还是软件系统，都需要“调度优化”。所谓“调度”，是指为了让系统的运行效率最优而采取的一系列方法。

在多任务系统中，合理的调度至关重要。在同样的硬件配置下，有良好调度方案的系统有时可以比调度不合理的系统，性能高出几十倍。在实际编程中，为了写出效率更高的程序，则必须更加贴近操作系统，因此有必要掌握线程与进程的调度关系。

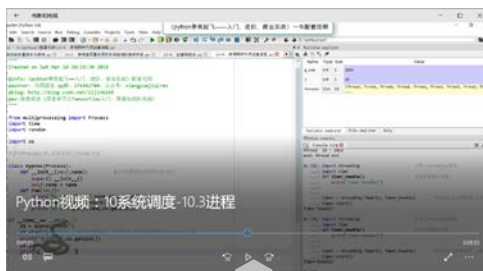
本章讲解使用 Python 开发“系统调度”相关的程序，主要内容是围绕着线程与进程展开的。

● 本章教学视频说明 ●

作者按照图书的内容和结构，录制了同步对应的教学视频。



按照图书的结构，像老师一样讲解



具体代码操作演示



Python视频：
10系统调度
-10.1线程与进
程10.2线程....



Python视频：
10系统调度
-10.3进程
mp4



Python视频：
10系统调度
-10.4协程
mp4

本章共有 3 段教学视频，总时长为 26 min 左右。包含下内容：

- 讲述了进程与线程的概念，及线程的创建及原理。接着又讲述了多线程的同步机制，演示了互斥锁的实现。最后又介绍了信号量、基于事件机制的消息队列、条件锁、定时器、线程池的相关知识。
- 讲述了进程的概念，并演示了一个实现进程的实例。
- 讲述了协程的概念，并演示了一个实现协程的实例。

10.1 进程与线程

进程，是应用程序的执行实例（比如 Windows 下某个运行起来的 exe 软件）。

线程，是执行进程中的路径。线程是进程的一部分。

线程主抓中央处理器执行代码的过程，其余的资源的管理和保护由进程去完成。

一个进程可以由一个或一个以上的线程组成。

线程与进程，又可以理解为对 CPU 时间段的描述：线程是 CPU 调度的最小单位，进程是 CPU 资源分配的最小单位。

操作系统通过线程与进程，可以将资源分配和调度分开，以实现更合理地使用 CPU 运算资源。例如：

- 在一个拥有两个 CPU 芯片的计算机上，进行批处理任务时，可以使用两个线程来完成批处理中最耗资源的工作（即每个 CPU 上各运行一个线程），从而达到机器资源的最佳应用。
- 如果程序中有人机交互功能，可以单独为交互服务启用一个线程。这样，在用户输入时，该线程可以暂停等待，而其他线程还可以继续工作。

10.2 线程

当一个进程里只有一个线程时，叫作单线程。超过一个线程就叫作多线程。

在多线程中，会有一个主线程来完成整个进程从开始到结束的全部操作，而其他的线程会在主线程的运行过程中被创建或退出。

10.2.1 线程的创建及原理

Python 中有关线程开发的部分，被单独封装到一个模块中。下面就来介绍一下模块的引入与使用。

1. 线程相关的模块

在 Python 中，提供了两个关于线程的模块。

- `_thread`：是 Python 3 以前版本中 `thread` 模块的重命名。提供了低级别的、原始的线程，以及一个简单的锁。功能比较有限。
- `threading`：Python 3 之后的线程模块，推荐使用。

这里以 `threading` 模块为例进行讲解。

2. 主线程的产生

一个 Python 程序就是一个进程。在运行 Python 程序时，该进程会默认启动一个线程，即主线程。可以通过 `threading` 模块中的 `current_thread` 函数，来查看当前线程的实例。例如：

```
import threading          #导入线程模块
print(threading.current_thread()) #对当前线程的查看，输出：<_MainThread(MainThread, started 6552)>
```

上面代码将 `current_thread` 函数的返回值打印出来，输出：`<_MainThread(MainThread, started 6552)>`。这个信息就是主线程 `MainThread` 的内容。其中，`MainThread` 表明这是一个主线程，而下面的 `started 6552` 是这个主线程的 ID 号。在操作系统中，每个线程都会有一个 ID 号，用来作为该线程的唯一标识。

`Threading` 模块中，除了 `current_thread`，还有两个常用函数。

- `threading.enumerate`：返回一个正运行线程的 list。“正运行”是指线程处于“启动后，且在结束前”状态，不包括“启动前”和“终止后”状态。
- `threading.activeCount`：返回正在运行的线程数量。与 `len(threading.enumerate())` 有相同的结果。

Python 中所有进程的主线程，名字都是一样的，即 `MainThread`。而子线程的名字需要在创建时指定。如不指定子线程的名字，Python 会为子线程添加默认名字。具体见下面子线程的创建方法。

3. 创建子线程

创建子线程有两种方法，都是通过 `threading.Thread` 类来实现的。具体如下：

- 直接对类 `threading.Thread` 进行实例化，并调用实例化对象的 `start` 方法创建线程。
- 用 `threading.Thread` 派生出一个新的子类，将新建类实例化，并调用其 `start` 方法创建线程。

两种方法大同小异，先来演示第一种方法。具体代码如下：

```
import threading

def handle(sid):
    #线程处理函数,内部打印出当前线程的参数及名称
    print("Thread %d run"%sid,threading.current_thread())

for i in range(1, 11):
    #使用一个循环来创建多个线程
    t = threading.Thread(target=handle, args=(i,)) #每次将计数器作为参数传入新建的线程中
    t.start() #启动线程
print("main thread",threading.current_thread()) #打印主线程信息
```

上面代码中，使用了一个 `for` 循环，来进行 `threading.Thread` 的多次实例化，并将 `handle` 函数作为每次实例化的参数，赋值给形参 `target`。线程对象在启动过程中，会先运行函数 `handle` 中的内容，再调用 `start` 方法，完成进行线程的启动。程序运行后，得到如下输出：

```
Thread 1 run <Thread(Thread-27, started 4480)>
Thread 2 run <Thread(Thread-28, started 7504)>
Thread 3 run <Thread(Thread-29, started 5336)>
Thread 4 run <Thread(Thread-30, started 2172)>
Thread 5 run <Thread(Thread-31, started 5640)>
Thread 6 run <Thread(Thread-32, started 7568)>
Thread 7 run <Thread(Thread-33, started 5664)>
Thread 8 run <Thread(Thread-34, started 7816)>
Thread 9 run <Thread(Thread-35, started 5108)>
Thread 10 run <Thread(Thread-36, started 7300)>
main thread <_MainThread(MainThread, started 6552)>
```

前 10 行的内容都是 `handle` 函数输出的。输出的结果中，每行开头单词 `Thread` 右面的数字为 `for` 循环中的计数器。该数字是在实例化线程对象时传入的。数字之后是每个子线程的信息。可以看到，系统为每个子线程分配了不同的默认名字（如：`Thread-27`、`Thread-28` 等）及唯一标识（如 4480、7504 等）。最后一行就是主线程的信息。

4. 定义 threading.Thread 类

threading.Thread 模块的初始化定义及解释如下：

```
class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *,
                        daemon=None)
```

具体的参数说明如下。

- **group**: 保留位，默认为空。为以后实现 ThreadGroup（线程组）类而预留的。
- **target**: 线程要执行的函数。如果为空，则意味着没有东西可以被调用。
- **name**: 线程的名称。默认是一个类似于 Thread-N 形式的名字，N 是十进制数字。
- **args**: 元组类型。表示在调用 target 时传入的参数。
- **kwargs**={}: 字典类型，即在调用 target 时传入的参数。
- **daemon**: bool 类型，为 True 时，表明该线程为守护线程，否则为 False。这个值必须与主线程的身份一致。即，如果主线程不是守护线程，所有的子线程都得被设为 False，否则会报 RuntimeError 异常。

守护线程指的是，在进程退出时，不用等待这个线程退出。除了上述的方法（即，在创建线程时直接指定线程的 daemon 属性），还可以使用 setDaemon 函数来指定守护线程，但是所有的设置都需要在调用 start 方法之前进行。实际编程中，守护线程不是很常用，读者了解一下即可。

5. threading.Thread 类的方法

了解完 threading.Thread 类的实例化参数，再看看该类所提供的方法。

- **run**: 线程活动的方法。
- **start**: 启动线程活动。
- **join**: 该方法有一个可选参数（timeout），用于阻塞当前上下文环境的线程，直到调用此方法的线程终止，或到达指定的 timeout（可选参数）。
- **isAlive**: 返回线程是否为活动的。
- **getName**: 返回线程名。
- **setName**: 设置线程名。

6. 线程的内部状态及原理

线程分为 5 种状态：创建、就绪、运行、阻塞和退出。其内部执行过程如下：

- (1) 在对 `threading.Thread` 实例化后，就完成了线程的创建。
- (2) 当程序调用 `start` 函数时，线程就会进入就绪状态，等待 CPU 分配时间片。
- (3) 如果分到时间片，则线程进入运行状态，执行 `run` 函数。
- (4) 在 `run` 函数执行期间，线程可以被打断，进入阻塞状态。
- (5) 如果阻塞结束，线程又会回到就绪状态，接着运行。
- (6) 运行结束（中途退出运行也算作运行结束），线程进入退出状态。

7. 重载 run

要创建并运行一个线程，必须先调用 `start` 方法将其启动；再调用线程的 `run` 方法，使线程真正运行起来。以线程类的方式创建线程时，可以在子线程类中重载 `run` 方法，来实现线程的处理流程。代码如下：

```
import threading
def handle(sid):                                #线程处理函数
    print("Thread %d run"%sid,threading.current_thread())
class MyThread(threading.Thread):               #定义 threading.Thread 的派生类
    def __init__(self,sid):                     #重载初始化方法
        threading.Thread.__init__(self)
        self.sid = sid
    def run(self):                             #重载 run 方法
        handle(self.sid)

for i in range(1, 11):                         #循环创建多线程
    t = MyThread(i)
    t.start()
```

上面代码中，定义 `threading.Thread` 的派生类 `MyThread`。在 `MyThread` 中重载了 `run` 方法，并在 `run` 方法中调用了 `handle` 函数，实现了线程处理函数的绑定。代码运行后，可以得到同样的结果。如下：

```
Thread 1 run <MyThread(Thread-97, started 6828)>
Thread 2 run <MyThread(Thread-98, started 2084)>
Thread 3 run <MyThread(Thread-99, started 3840)>
Thread 4 run <MyThread(Thread-100, started 3572)>
Thread 5 run <MyThread(Thread-101, started 6884)>
Thread 6 run <MyThread(Thread-102, started 4708)>
Thread 7 run <MyThread(Thread-103, started 7424)>
```

```
Thread 8 run <MyThread(Thread-104, started 6856)>
Thread 9 run <MyThread(Thread-105, started 6048)>
Thread 10 run <MyThread(Thread-106, started 8172)>
```

程序输出的内容同样都是来自 `handle` 函数。这表明，在 `for` 循环里使用 `MyThread` 类创建的线程都执行了 `handle` 函数。

总结一下。将处理函数绑定到线程上的方法，也有两种：

- 在实例化时，为其 `target` 参数赋值。
- 继承 `threading.Thread` 类，重载其 `run` 函数，并在其中指定。

两种方法没有好坏之分。在应用时，可以根据整体代码的风格，选择适合的方式，做到活学活用。

10.2.2 互斥锁

多线程的优势在于并发性，即，可以同时运行多个任务。但是当线程需要共享数据时，也可能会带来数据不同步的问题。互斥锁的作用就是解决数据不同步问题。下面通过例子复现数据不同步问题，并进行解决。

1. 多线程的问题

假设需要创建 10 个线程。要求：每创建一个线程，就对公共计数器乘以 2；同时这个公共计数器当时的值还要与线程的顺序值对应。

代码如下：

```
import threading
import time

g_num = 1                                #公共计数器
def handle(arg):                          #线程处理函数
    global g_num
    g_num *=2                             #计数器乘以 2
    time.sleep(arg%2)                    #类似与其他操作，这里用 sleep 代替计算时间
    print ("thread ",arg,":",g_num)      #输出线程对应的计数器数值

threads = []                              #建立线程列表

for i in range(1,11):                    #循环创建线程
```

```

t = threading.Thread(target=handle, args=(i,))
t.start()
threads.append(t)                                #将创建的线程全都放到列表里

for t in threads:                                #等待所有线程完成
    t.join()

print ('main thread end')                        #程序结束，退出主线程

```

首先，代码中定义了一个公共计数器 `g_num`。接着，在线程处理函数中使用 `time.sleep` 函数让程序休眠。并将公共计数器乘以 2，同时输出线程对应的计数器的值。

上面代码中倒数第 3 行，使用了 `join` 函数。`join` 函数执行时，会将当前主线程暂停，等待子线程 `t` 运行结束。子线程 `t` 运行结束后，主线程才继续恢复执行。整个程序运行后，得到如下结果：

```

01 thread 2 : 4
02 thread 4 : 16
03 thread 6 : 64
04 thread 8 : 256
05 thread 10 : 1024
06 thread 1 : 1024
07 thread 3 : 1024
08 thread 5 : 1024
09 thread 7 : 1024
10 thread 9 : 1024
11 main thread end

```

可以看到，程序的输出中，线程的顺序与计数器的值完全对不上（见第 6 行，1 号线程的计数器变成了 1024）。这不符合需求。造成这种现象的原因是：当一个子线程调用 `sleep` 进行休眠时，其他线程会接着修改原有的公共计数器。这导致，休眠结束后取得的那个公共计数器的值与原来不一样了。这就是数据的不同步。为了避免这种情况，引入了“锁”的概念。

2. 互斥锁

锁的出现，主要就是为了解决多线程间同步的问题。其概念的核心意思在于：将执行程序中的某段代码保护起来（相当于锁起来），被锁起来的代码一次只能允许一个线程执行。利用这个概念，“将上面的公共计数器乘以 2”与输出公共计数器的操作锁起来，就可以实现多线程的同步了。

在 Python 中，使用 `threading.RLock` 类来创建锁。`threading.RLock` 类有两个方法——`acquire` 与 `release`。

- `acquire` 负责开始对代码进行保护。在 `acquire` 之后的代码，都将只允许一个线程进行执行。
- `release` 方法用于停止保护（即释放锁资源）。在 `release` 之后的代码又恢复到原来的样子，可以被多线程交叉执行。

下面通过代码来为前面的代码加上锁，进行数据同步的修复：

```

01 import threading
02 import time
03
04 g_num = 1                                #公共计数器
05 lock = threading.RLock()                 #锁的实例化
06 def handle(arg):                         #线程处理函数
07     lock.acquire()                       #加锁
08     global g_num
09     g_num *=2
10     time.sleep(arg%2)
11     print ("thread ",arg,":",g_num)
12     lock.release()                       #解锁
13
14 threads = []
15
16 for i in range(1,11):
17     t = threading.Thread(target=handle, args=(i,))
18     t.start()
19     threads.append(t)
20
21 # 等待所有线程完成
22 for t in threads:
23     t.join()
24
25 print ('main thread end')
```

如上面代码中第 5 行，在函数 `handle` 的外部实例化了 `threading.RLock` 类，得到了变量 `lock`。并在函数 `handle` 的内部，“对 `g_num` 乘 2”操作与输出操作的两端进行了加锁（第 7 行）和解锁（第 12 行）。代码运行后，输出如下结果：

```

thread 1 : 2
thread 2 : 4
```

```
thread 3 : 8
thread 4 : 16
thread 5 : 32
thread 6 : 64
thread 7 : 128
thread 8 : 256
thread 9 : 512
thread 10 : 1024
main thread end
```

可以看到，由于使用了锁保护，线程的序号与“公共计数器乘 2”的顺序完全对应上了，从而实现了多线程的同步。

其实在操作系统中，会存在着各种复杂的多线程并发情况，一般都是需要通过加锁来解决。锁的种类也会有很多种。这里介绍的是典型的互斥锁，也是最简单、常用的一种锁。类似的还有条件锁、全局锁等，后文会有详细介绍。



注意：

从系统的角度来看，锁的作用其实是将多线程变回到了单线程。这是以牺牲性能，来换取程序的准确性。在代码设计中，应该最大化地避免使用锁。即使加了锁，也要让被保护的区域尽量地少，在满足准确性的同时实现性能最大化。在代码中，有“加锁”操作，就一定要有与之对应的“解锁”操作，否则代码将失去多线程的优势。

10.2.3 实例 31：使用信号量来同步多线程间的顺序关系

信号量（semaphore）是一种带计数的线程同步机制。调用 `release` 函数时，计数器加 1；调用 `acquire` 函数时，计数器减 1。当计数为 0 时，线程会自动阻塞，等待 `release` 被调用。

在 Python 中存在两种信号量：一种是纯粹的信号量（`Semaphore`），另一种是带有边界的信号量（`BoundedSemaphore`）。二者的区别如下。

- **Semaphore**：在调用 `release` 函数时，单纯地将计数器加 1。不会去检查加 1 后的计数器是否超过上限。
- **BoundedSemaphore**：在调用 `release` 函数时，会去检查计数器加 1 后是否超过上限。对计数器的上限进行校验。是一个更加安全的机制。

在实际情况中，希望多线程以一定的先后顺序来执行。例如：生产者与消费者的关系，一定是先生产后消费。下面就使用信号量来实现多线程之间的关系同步。

实例描述

先定义两个线程处理函数：一个是生产者，一个是消费者。使用信号量来同步两个函数顺序关系。在生产者函数中，不定期的完成生产任务。消费者函数中直接进行消费操作。

接着，同时启用若干个关于生产者及消费者的线程。观察程序运行情况。

在多线程环境下，线程数量与系统的资源数量相匹配，才会使整体性能最优。信号量的作用就在于此。系统通过信号量来维护着一个计数器。该计数器代表了，可同时访问资源或者进入临界区的线程数。有了信号量的限制机制，整个软件系统的性能更加稳定。

1. 信号量实现

信号量，是通过 `threading` 中的 `Semaphore` 方法来实现的（边界信号量是通过 `BoundedSemaphore` 方法来实现的）。使用时，需要将信号量的个数传入即可。下面分别用例子来演示具体用法。

2. 代码实现

在代码中，定义了一个全局变量 `item`。生产者生产出的商品编号会被放到 `item` 中。消费者处理函数中，会将 `item` 打印出来，以代表该商品已经被消费掉。具体代码如下：

代码 10-1：使用信号量来同步多线程间的顺序关系

```
import threading
import time
import random

semaphore = threading.Semaphore(0)           #创建信号量

def consumer():                                #定义消费者
    print("consumer :挂起.")
    semaphore.acquire()
    print("consumer : 消费 %s." % item)

def producer():                                #定义生产者
    global item                                #定义商品编号
    time.sleep(3)
    item = random.randint(1, 1000)           #产生随机数
    print("producer : 生产 %s." % item)
    semaphore.release()
```

```

threads = []                                #定义列表收集线程

for i in range(0, 2):                        #使用循环完成生产者与消费者线程的建立
    t1 = threading.Thread(target=producer)
    t2 = threading.Thread(target=consumer)
    t1.start()
    t2.start()
    threads.append(t1)
    threads.append(t2)

for t in threads:                            #等待所有线程完成
    t.join()

```

上述代码中，在生产者线程处理函数中，使用 `sleep` 函数休眠了 3 秒钟，来模拟生产过程所用的时间。在消费者线程处理函数中，通过 `semaphore.acquire` 方法挂起线程，等待信号量的通知。代码运行后，生成如下结果：

```

consumer : 挂起.
consumer : 挂起.
producer : 生产 807.
consumer : 消费 807.
producer : 生产 420.
consumer : 消费 420.

```

前两行输出了“consumer:挂起”，这表示线程刚启动时，消费者线程进入到“挂起”状态。生产者使用 `random.randint` 函数生成随机数，并赋值给全局变量 `item`。在后面的 4 行输出中，显示出生成者生成的编号（807、420）及消费者的消费信息。

3. 使用边界信号量

如果要使用边界信号量 `BoundedSemaphore`，将第 5 行代码换成调用 `BoundedSemaphore` 即可。例如：

```
semaphore = threading. BoundedSemaphore (1)
```

这里的参数不能设为 0。否则会报如下异常错误：

```
ValueError: Semaphore released too many times
```

这个错误表明，信号量的上限超过了上限设置值 0。另外，`BoundedSemaphore` 的起始值设为了 1，表明第一个消费者线程是可以消费 `item` 的。因为这时没有 `item` 的定义，所以还需要将全局变量 `item` 的设置放在外部，来保证程序运行不会报错。同时为了保证程序内容的准确性，

可以给 item 初始值一个在正常编号（1~1000 以外的数）。然后对 item 的取值进行判断，过滤掉初始值。

10.2.4 实例 32：实现基于事件机制的消息队列

事件机制，是线程间最简单的通信机制。它使得线程或是进程间的同步，可以通过事件告知的方式来实现。在 Windows 操作系统中，消息通知机制也是以事件的方式来实现的。下面通过一个实例来演示时间机制。

实例描述

创建两个线程，让其共同操作一个队列。一个线程是 `readthread`，负责从队列里往外读取数据。一个线程是 `writethread`，负责写入数据。

要求：`readthread` 读时，`writethread` 不能写；`writethread` 写时，`readthread` 不能读。

这个案例涉及到了事件操作的知识，下面先来介绍一下。

1. 事件实现

Python 中，事件的实现使用了 `threading` 模块中的 `Event` 类。该类有三个方法。

- `set`：设置事件。将标志位设为 `True`。
- `wait`：等待事件。会将当前线程阻塞，直到标志位变为 `True`。
- `clear`：清除事件。将标志位设为 `False`。

事件，可以通过 `Event` 类的实例化来实现。可以使用上面这 3 个方法，来实现整个事件的生命周期。

2. 代码实现

实现两个线程类 `WriteThread`、`ReadThread`，分别用于队列的写和读。两个线程类的初始化函数传入共同操作的队列（q）、读事件（RE）、写事件（WE）。

- 在写队列类（`WriteThread`）中，随机生成 5 个 1~10 之间的数并传入队列，并进行对读事件（RE）的通知，之后等待写事件的到来。
- 在读队列类（`ReadThread`）中，等待读事件（RE）的通知。当收到时间后，开始读队列，并通知写事件（WE）。

代码如下：

代码 10-2：基于事件机制的消息队列

```
import queue                                #引入队列模块
from random import randint
from threading import Thread
from threading import Event

class WriteThread(Thread):                  #写数据线程
    def __init__(self,q,WE,RE):
        super().__init__()
        self.queue = q
        self.RE = RE
        self.WE = WE
    def run(self):
        data = [randint(1,10) for _ in range(0,5)]
        self.queue.put(data)
        print("WriteThread 写队列: ",data)
        self.RE.set()                       #通知读线程可以读了
        print("WriteThread 通知读事件")
        print("WriteThread 等待写事件")
        self.WE.wait()                      #等待写事件
        print("WriteThread 收到写事件")
        self.WE.clear()                     #清除写事件，以方便下次读取

class ReadThread(Thread):                   #读数据线程
    def __init__(self,q,WE,RE):
        super().__init__()
        self.queue = q
        self.RE = RE
        self.WE = WE
    def run(self):
        while True:
            self.RE.wait()                   #等待读事件
            print("ReadThread 收到读事件")
            data = self.queue.get()
            print("ReadThread 读队列 {0}".format(data))
            print("ReadThread 发送写事件")
            self.WE.set()                     #发送写事件
            self.RE.clear()                   #清除读事件，以方便下次读取

q= queue.Queue()
WE = Event()
```

```

RE = Event()
writethread = WriteThread( q, WE, RE)           #实例化线程类
readthread = ReadThread(q, WE, RE)

writethread.start()                             #启动线程
readthread.start()

```

建好两个类之后，实例化队列、读事件、写事件，并将这三个对象传入读写线程 `ReadThread` 与 `WriteThread` 中，进行实例化。最终使用线程的 `start` 方法启动。代码运行后，输出如下结果：

```

WriteThread 写队列: [4, 4, 4, 9, 10]
WriteThread 通知读事件
WriteThread 等待写事件
ReadThread 收到读事件
ReadThread 读队列 [4, 4, 4, 9, 10]
ReadThread 发送写事件
WriteThread 收到写事件

```

线程启动之后，执行的过程如下：

(1) 写线程 `writethread` 启动后，在类 `WriteThread` 中，进行写队列操作。输出了第 1 行的结果：“`WriteThread 写队列: [4, 4, 4, 9, 10]`”。

(2) 读线程 `readthread` 启动后，在类 `ReadThread` 中，使用事件 `RE` 的 `wait` 方法等待该事件，线程一直处在挂起状态。

(3) 写线程 `writethread` 做完写入队列操作之后，通过事件 `RE` 的 `set` 方法，发出一条可读事件。输出了第 2 行结果：“`WriteThread 通知读事件`”。

(4) 程序挂起，等待写事件的到来。输出了第 3 行结果：“`WriteThread 等待写事件`”。

(5) 在第 3 步之后，读线程收到了 `WriteThread` 中发出的读事件，开始打印。输出了第 4 行结果：“`ReadThread 收到读事件`”。

(6) 读线程类 `ReadThread` 开始进行读队列操作。输出了第 5 行结果：“`ReadThread 读队列 [4, 4, 4, 9, 10]`”。

(7) 发出写事件，通知写线程类 `WriteThread`。输出了第 6 行结果：“`ReadThread 发送写事件`”。

(8) 在读线程类 `ReadThread` 中，收到写事件，并打印。输出了最后一行结果：“`WriteThread 收到写事件`”。

10.2.5 实例 33：使用条件锁同步多线程中的生产者与消费者关系

条件锁，是线程同步中更高级的一种应用。它是在保护互斥资源的基础之上，又增加了条件判断的机制。在判断当前情况不满足某条件时，条件锁可以主动挂起当前程序，等待条件准备好后再去执行。

实例描述

定义两个线程处理函数：一个是生产者，另一个是消费者，并使用条件锁进行顺序关系的同步。在生产者函数中，不定期的完成生产任务。在消费者函数中，直接进行消费操作。

接着，同时启用若干个关于生产者及消费者的线程。观察程序运行情况。

本例子是在 10.1.4 小节的实例基础上进行修改，实现更复杂的逻辑功能。即，在生产者与消费者之间又增加了一个条件关系的规则。

1. 条件锁实现

Python 中事件的实现，使用了 `threading` 模块中的 `Condition` 类。它也有 `acquire` 方法和 `release` 方法，而且还有 `wait`、`notify` 和 `notifyAll` 方法。这三个方法的说明如下。

- `wait`：挂起当前线程，等待唤起。
- `notify`：唤起被 `wait` 函数挂起的线程。
- `notifyAll`：唤起所有线程，防止线程永远处于沉默状态。

2. 代码实现

在代码中，定义了一个全局变量 `itemNum`，作为生成者产出的商品个数。

- 生产者处理函数每生成一个商品，就将 `itemNum` 加 1，同时发送条件锁通知。
- 消费者处理函数会判断 `itemNum` 变量。如果商品个数为 0，则挂起程序等待。当收到条件锁通知后，则打印出来代表消费该商品，同时将 `itemNum` 减 1。

具体代码如下：

代码 10-3：条件锁的应用

```
from threading import Thread
from threading import Condition
import time
import random
```



```

c = Condition()                                #创建条件锁
itemNum = 0
item = 0

def consumer():                                #定义消费者
    global item                                #定义商品编号
    global itemNum                             #定义商品个数
    c.acquire()                                #锁住资源
    while 0==itemNum:                           #如无产品则让线程等待
        print("consumer :挂起.")
        c.wait()
    itemNum -=1
    print("consumer : 消费 %s." % item,itemNum)
    c.release()                                #解锁资源

def producer():                                #定义生产者
    global item                                #定义商品编号
    global itemNum
    time.sleep(3)
    c.acquire()                                #锁住资源
    item = random.randint(1, 1000)
    itemNum +=1
    print("producer : 生产 %s." % item)
    c.notifyAll()                              #唤醒所有等待的线程（消费者）
    c.release()                                #解锁资源

threads = []                                  #定义线程收集列表

for i in range(0, 2):                          #使用循环完成生产者与消费者线程的建立
    t1 = Thread(target=producer)
    t2 = Thread(target=consumer)
    t1.start()
    t2.start()
    threads.append(t1)
    threads.append(t2)

for t in threads:                              #等待所有线程完成
    t.join()

```

在生产者线程处理函数中，先使用条件锁的 `acquire` 函数占用资源，接着生成了一个商品编号 `item`，并且对计数器 `itemNum` 加 1。然后，使用条件锁的 `notifyAll` 函数将挂起状态的消费者线程 `t2` 唤醒，在消费者线程处理函数 `consumer` 中将商品编号打印出来。代码运行后，生成如下结果：

```

consumer :挂起.
consumer :挂起.
producer : 生产 966.
consumer : 消费 966. 0
consumer :挂起.
producer : 生产 270.
consumer : 消费 270. 0

```

上述结果中，第 3 行显示了“producer：生产 966”内容，表明生产者线程所生产商品的编号为 966。之后，使用条件锁的 `notifyAll` 函数唤醒挂起的消费者线程，这时两个消费者线程都被唤醒：

- 一个消费者线程取到商品，并将其输出（见第 4 行显示）。接着完成整个过程，该线程退出。
- 另一个消费者线程发现仍然没有商品，于是接着挂起（见第 5 行显示）。

直到第二个生产者线程又生产了一个商品（见第 6 行显示），将两个消费者线程唤醒（见第 7 行显示）。

10.2.6 实例 34：创建定时器触发程序，在屏幕上输出消息

定时器，是 Python 应用在运维自动化项目中的常用方法。在其他方向的项目中也有广泛的应用。其功能是设定一个时间，当到达该时间时让机器自动去做某个事情。这种方式又叫作“时间触发事件”。

实例描述

创建一个定时器，并实现如下功能：

- （1）一秒钟之后，在屏幕打印一句话：“Timer headle!”。
 - （2）之后每隔一秒在屏幕上打印一次：“Timer headle!”。
 - （3）为第二步加个限制，只允许持续 10 秒。
-

先创建定时器，再使用代码完成定时器单次触发、循环触发等操作。

1. 创建定时器

Python 中的定时器，是使用 `threading` 模块下的 `Timer` 类来实现的。它本质上还是线程。只不过多了一个时间参数。定义如下：

```
class threading.Timer(interval, function, args=[], kwargs={})
```

参数的说明如下。

- **interval**: 触发定时器的时间。
- **function**: 定时器到时间后的处理函数。
- 其他: 传入处理函数的参数。

需要注意的是，定时器的处理函数是可以传入参数的。定时器在实例化之后，便可以像线程一样通过 `start` 方法来运行。

2. 单次触发定时器

下面使用函数 `timer_headle` 作为定时器的触发函数。在实例化 `threading.Timer` 时，传入的参数 `interval` 为 1，代表将“定时间隔”设为一秒。代码如下：

代码 10-4：创建定时器触发程序在屏幕上输出消息

```
01 import threading                #导入 threading 模块
02 import time
03 def timer_headle():              #定时器触发函数
04     print('Timer headle!')
05
06 timer = threading.Timer(1, timer_headle)    #实例化定时器线程，1 秒后执行线程处理函数
07 timer.start()
```

上面代码运行之后，没有任何反应。一秒钟之后，输出如下结果：

```
Timer headle!
```

这说明，在一秒钟之后触发了定时器事件，系统调用了 `timer_headle` 函数，于是在屏幕上输出了信息。

3. 循环触发定时器

在定时器的触发函数里，再创建一个定时器，即可实现定时器的循环触发。

下面代码中，在定时器触发函数 `loop_timer_headle` 中，又创建了一个定时器，并将处理函数指向自己。这样，每次执行完当前触发函数后，又创建一个新的定时器，源源不断地调用 `loop_timer_headle` 函数在屏幕上打印信息。

代码 10-4：创建定时器触发程序在屏幕上输出消息（续）

```

08 def loop_timer_headle():                                #定时器循环触发函数
09     print('Timer headle!')
10     global timer2
11     timer2 = threading.Timer(1, loop_timer_headle)      #创建定时器
12     timer2.start()
13
14 timer2 = threading.Timer(1, loop_timer_headle)
15 timer2.start()

```

代码运行后显示如下：

```

Timer headle!
Timer headle!
.....

```

每隔一秒钟就会输出一句“Timer headle!”的消息。程序在 `loop_timer_headle` 函数中又创建了一个新的定时器，新的定时器又会触发 `loop_timer_headle` 的调用。这样一直循环下去。可以通过关闭 console 的方式结束本次执行。

4. 为循环触发定时器加上时间限制

在 `threading.Timer` 中，可以使用 `cancel` 方法来退出定时器。

在下面代码中，使用 `sleep` 函数让主线程休眠 10 秒，之后调用 `cancel` 结束定时器。

代码 10-4：创建定时器触发程序在屏幕上输出消息（续）

```

16 time.sleep(10)                                          #休眠10秒
17 timer2.cancel()                                        #结束定时器

```

上面代码运行后，程序会显示 10 秒钟信息，接着停止显示。



注意：

这里演示的定时器传入时间是 1 秒。还可以通过传入浮点型数值，来实现更小精度的定时功能。例如：1.5，表明让定时器在 1.5 秒之后触发。

10.2.7 实例 35：使用线程池提升运行效率

线程池是一种多线程处理形式，是在正常的多线程处理方式上的一种优化。

- 正常线程使用方式是：创建线程，启用，结束，销毁。

- 线程池的处理方式是：在程序启动时就创建好若干个线程，并保存到内存中。当线程启动并执行完成之后，并不做销毁处理，而是等待下次再使用。这样内存中的线程就像个池子一样，需要用时过来取，用完了再还回去。

在需要频繁创建线程的系统中，一般都会使用线程池技术。原因有两点：

- 每一个线程的创建，都是需要占用系统资源的，是一件相对耗时的事情。同样，在销毁线程时，还需要回收线程资源。线程池技术，可以省去创建与回收过程中所浪费的系统开销。
- 在某些系统中，需要为每个子任务来创建对应的线程（例如爬虫系统中的子链接）。这种情况会导致线程数量失控性暴涨，直到程序崩溃。线程池技术可以很好地固定线程的数量，保持程序稳定。

下面演示具体案例。

实例描述

编写代码，实现一个列表，其中的内容是具体的人名；编写函数，输出传入内容；调用函数，依次处理列表中的元素。并做如下操作：

- （1）使用一个主线程进行处理并记录处理时间。
 - （2）使用线程池处理并记录处理时间（两种方式：抢占式、顺序式）。
 - （3）对两个时间进行比较，体会多线程的优势。
-

先介绍线程池的基本实现，接着按照实验步骤，依次编写单线程处理、多线程处理的代码。

1. 实现线程池

Python 中，使用 `concurrent.futures` 模块下的 `ThreadPoolExecutor` 类来实现线程池。在实例化时，会将需要的线程个数传入。系统就会为该线程池初始化相应个数的线程。线程池的使用有两种方式。

- 抢占式：线程池中的线程执行顺序不固定。该方式使用 `ThreadPoolExecutor` 的 `submit` 方法实现。
- 非抢占式：线程将按照调用的顺序执行。此方式使用 `ThreadPoolExecutor` 的 `map` 方法来实现。

从使用角度来看：抢占式更灵活；非抢占式更严格。

- 抢占式，允许池中线程的处理函数不一样。如执行过程中某个线程出现异常，也不影响其他线程。
- 非抢占式，要求线程池中的线程必须执行同样的处理函数。而且，一旦某个线程出现异常，其他线程也会停止。

2. 实现单线程处理程序，并记录时间

代码中，使用了 `time` 模块记录程序运行的时长。具体方法为：在运行之前获得当前时间 `start1`，运行之后再次获得当前时间 `end1`。最终两个时间的差值，就是该线程的运行时间。

为了让案例的两种情况比较起来更明显，在线程处理函数 `printperson` 中，调用 `sleep` 使程序休眠两秒钟，将程序的运行时间差变大。具体代码如下：

代码 10-5：创建线程池

```
01 from concurrent.futures import ThreadPoolExecutor
02 import time
03
04 def printperson(p):                                #定义线程池处理函数
05     print(p)
06     time.sleep(2)
07
08 person=["Anna","Gary","All"]
09 start1=time.time()                                #获取开始时间
10 for p in person:                                   #调用函数，将列表元素依次传入
11     printperson(p)
12 end1=time.time()                                    #获取开始时间
13 print("time1: "+str(end1-start1))                  #打印时间差
```

上面代码运行后，输出如下结果：

```
Anna
Gary
All
time1: 6.0158867835998535
```

结果显示，整个程序运行所用时间为 6 秒多。因为是单线程按顺序执行，所需的时间是每次函数 `printperson` 运行的总和。

3. 实现抢占线程池

使用 `with` 关键字来创建线程池，调用实例化对象的 `submit` 方法将线程启用。代码如下：

代码 10-5：创建线程池（续）

```
14 start2=time.time()
15 with ThreadPoolExecutor(3) as executor:      #创建抢占式线程池，池内有3个线程
16     for p in person:
17         executor.submit(printperson,p)      #启用线程
18 end2=time.time()
19 print("time2: "+str(end2-start2))
```

上面代码运行后，输出如下结果：

```
Anna
Gary
All
time2: 2.00089430809021
```

结果显示，整个程序运行所用的时间是两秒多，比单线程的程序所用的运行时间（6 秒多）少很多。这是由于，多线程中的并发机制缩短了运行程序所需要的时间。

4. 实现非抢占线程池

非抢占线程池也是使用 `with` 关键字来创建的。不同的是，非抢占线程池是通过调用实例化对象的 `map` 方法来启用线程。代码如下：

代码 10-5：创建线程池（续）

```
20 start3=time.time()
21 with ThreadPoolExecutor(3) as executor1:
22     executor1.map(printperson,person)
23 end3=time.time()
24 print("time3: "+str(end3-start3))
```

上面代码运行后，输出如下结果：

```
Anna
Gary
All
time3: 2.0017685890197754
```

结果显示，整个程序运行所用的时间是两秒多，与抢占式线程池的方法效率相差不多。这比单线程的程序运行所用时间少。

10.2.8 需要创建多少个线程才算合理

多线程的并发机制，虽然可以提升程序效率，但线程个数也不是越多越好。如要找到更优的线程数量，可以使用如下方法：

- (1) 初始化一定数量的线程。
- (2) 在多次实验中递增或递减线程数量，测试运行性能。
- (3) 确定最优的线程数量。

其中的第(1)步初始化线程的个数，可以先查看单个任务的 CPU 消耗，然后直接乘以百分比。而第(2)步，评估运行性能的方法，从外部观察每秒处理的任务数，算出批处理全部任务所用的时间。

另外，还可以通过公式计算得出线程数量的参考值：

$$\text{服务器端最佳线程数量} = \frac{\text{线程等待时间} + \text{线程 CPU 时间}}{\text{线程 CPU 时间}} \times \text{CPU 数量}$$

10.3 进程

进程，就是操作系统中具体的处理任务。每一个进程都会有自己独立的内存空间。它是线程的载体。

一般来讲，某一个程序启动时，就会默认启动一个进程，将该程序装载到内存。同时在该进程中还会默认启动一个线程，来执行本进程中的内容。

10.3.1 实例 36：创建多进程的程序

在 Python 中，进程相关的实现被统一封装在模块 `multiprocessing` 中。进程的创建方式有两种：

- 继承 `multiprocessing` 中的 `Process` 类，并重写 `run` 方法。
- 直接通过 `multiprocessing` 中的进程类 `Process` 进行创建。

类似于线程的创建，编写程序对 `Process` 类实例化时，直接将进程的处理函数传入即可。在 `Process` 类的实例化参数中，也是通过 `target` 来接收处理函数的。

创建好进程后，可以通过实例化对象的 `start` 方法将其启动。下面通过实例演示。

实例描述

分别使用继承类与类的实例化方式创建多线程程序，并做如下操作：

- (1) 向进程传入参数。
 - (2) 打印程序中已启动的进程 ID。
 - (3) 分析输出结果，理解多线程程序。
-

本实例将使用命令行的方式来运行 Python 程序。原因是，Spyder 的操作界面默认只支持单进程运行，无法看到多进程程序运行的效果。

1. 使用继承类的方式创建进程

实现自定义进程类 myproc，使其继承于系统进程类 Process。重写 myproc 类中的 run 方法，实现进程的逻辑。在 run 方法中，调用 sleep 函数让程序挂起一段时间。并调用 os 模块的 getpid 方法，得到当前进程的 ID，将其打印出来。类似于线程，进程的实例化对象也有一个 join 函数，功能也是挂起当前主进程，等待子进程结束。具体代码如下：

代码 10-6：使用两种方式创建进程

```
01 from multiprocessing import Process    #导入 multiprocessing 模块
02 import time
03 import random
04
05 import os
06
07 class myproc(Process):                  #继承 Process 类，并实现自己的 run 方法
08     def __init__(self,name):            #必须调用父类的 init 方法
09         super().__init__()
10         self.name = name
11     def run(self):
12         print("%s start"%self.name,os.getpid())
13         time.sleep(random.randint(1,3))
14         print("%s end"%self.name,os.getpid())
15
16 if __name__=='__main__':                #使用创建进程语句时，必须要有这句
17     p1 = myproc("test")                 #实例化子进程并传入参数
18     p1.start()                           #启动进程
19     print("主进程! ",os.getpid())
20     p1.join()
21     print ('主进程结束.')
```

在上面代码中，`myproc` 类在继承 `Process` 的同时，也重载了初始化函数。通过自定义初始化函数的参数，来实现子进程的传值。

代码完成后，保存成扩展名为 `py` 的文件。单击“开始”菜单，在运行窗口中输入“`cmd`”命令，进入控制台界面。通过 `DOS` 命令找到保存好的代码文件。执行如下命令：

```
D:\python2>python "10-6 使用两种方式创建进程.py"
```

之后显示结果如下：

```
主进程! 4488
test start 4588
test end 4588
主进程结束.
```

第一行和最后一行是主进程打印的内容，主进程的 ID 为 4488。中间两行是子进程的输出，子进程的 ID 为 4588。



注意：

在编写多进程程序时，创建进程的代码必须在指定当前模块为 `main` 的代码块内执行（见代码 10 行），否则会报运行时态的错误。

2. 使用类的实例化方式创建进程

定义进程处理函数 `run_proc`。在实例化 `Process` 时，将 `run_proc` 传入。接着使用实例化对象 `p` 的 `start` 方法启动进程。具体代码如下：

代码 10-6：使用两种方式创建进程（续）

```
22 def run_proc(name):                                     #进程处理函数
23     print("%s start"%name,os.getpid())
24     time.sleep(random.randint(1,3))
25     print("%s end"%name,os.getpid())
26
27 if __name__=='__main__':
28     p = Process(target=run_proc, args=('test',)) #实例化
29     print("主进程! ",os.getpid())
30     p.start()                                           #启动进程
31     p.join()
32     print ('主进程结束.')
```

将上面代码保存成扩展名为 py 的文件。单击“开始”菜单，在运行窗口中输入“cmd”命令，进入控制台界面。通过 DOS 命令找到保存好的代码文件。执行如下命令：

```
D:\python2>python "10-6 使用两种方式创建进程.py"
```

之后显示结果如下：

```
主进程! 1040
test start 5092
test end 5092
主进程结束。
```

结果中 1040 为主进程的 ID。test 为传入子进程的字符串，5092 为子进程的 ID。

3. 总结

上面两种方法都是创建进程的常用方法。在编写多进程程序时，进程的处理函数一旦出现问题，是很难调试的。一般常用的方法是：先使用单进程进行运行，跟踪代码；待程序验证正确之后，再将其改为多进程程序。

10.3.2 多进程与多线程的区别

多进程与多线程，都可以使用并行机制来提升系统的运行效率。二者的区别在于：运行时所占的内存分布不同。

- 多线程是共用一套内存的代码块区间。
- 多进程是各用一套独立的内存区间。

由于这个特性，常常会用多进程来实现守护服务器的功能。而多线程更适用与批处理任务等功能。

在大型的计算机集群系统中，都会将多进程程序分布运行在不同的计算机上协同工作。而每一台计算机上的进程内部，又会由多个线程来并行工作。

10.4 协程

协程（coroutine），可以理解为是线程的优化，有的地方有称之为轻量级进程。它是一种比线程更节省资源、效率更高的系统调度机制。它的特点是，在同时开启的多个任务中，一次只执行一个。如果当前任务遭遇阻塞，才会切换到下一个任务继续执行。这种机制可以实现多

任务的同步，又能够成功地避免线程中使用锁的复杂性，简化了开发。

10.4.1 协程的相关概念及实现步骤

早先的协程是使用生成器关键字 `yield` 来实现的，代码特别复杂难懂。自从 Python 3.5 之后，确定了协程的语法，使得创建协程的方式得到改善。

在 Python 中，能够实现协程的模块有多个，如 `asyncio`、`tornado` 或 `gevent`。

1. 协程的相关概念

这里以 `asyncio` 为例，先来了解一下创建协程所用到的概念。

- `event_loop`（事件循环）：是一个协程处理函数的调用机制。程序会开启一个无限循环，当事件发生时，调用相应的协程函数。
- `coroutine`（协程对象）：指一个使用 `async` 关键字来定义的函数。调用该函数，会返回一个协程对象。该协程对象就是一个处于挂起状态的协程函数，需要注册到事件循环 `event_loop` 中，由事件循环 `event_loop` 进行调用。
- `task` 任务：是对协程的进一步封装。
- `future`：等同于 `task`。代表执行任务的结果。
- `async/await` 关键字：Python 3.5 中有两个用于定义协程的关键字。`async` 用于定义一个协程，`await` 用于挂起阻塞的异步调用接口。

2. 协程的实现步骤

下面通过代码演示最基本的协程实现：

```
import asyncio                                #引入 asyncio 模块

async def do_some_work(x):                    #定义协程处理函数
    print( x)
coroutine = do_some_work('hello')            #生成协程对象，并传入 hello
loop = asyncio.get_event_loop()               #获得事件循环对象
try:
    loop.run_until_complete(coroutine)        #将协程注册到实现事件循环对象中，并开始运行。输出：
hello
finally:
    loop.close()                             #程序结束关闭事件循环对象
```

上面的代码体现了实现协程的基本步骤：

- (1) 使用 `async` 关键字定义协程处理函数；
- (2) 生成协程对象；
- (3) 调用 `asyncio` 中的 `get_event_loop` 函数获得事件循环对象；
- (4) 调用事件循环对象的 `run_until_complete` 方法，运行协程处理函数

10.4.2 实例 37：使用协程实现“任务提交”与“结果接收”

协程的特性是，按顺序执行、通过挂起的方式在并行任务切换。这些特征常会被用在批处理任务中。下面就演示一个利用协程实现任务提交与结果接收的例子。

实例描述

使用协程机制处理一个任务。要求：提交任务时传入任务参数，任务结束后获得执行结果。

本实例将使用任务来封装协程的处理函数，并在任务中通过回调的方式得到处理函数的结果。

1. 封装及回调协程的任务

在得到协程对象与事件循环对象后，有两种创建任务的方式：

- (1) 使用事件循环对象的 `create_task` 方法；
- (2) 使用 `asyncio` 模块下的 `ensure_future` 函数。

得到任务对象之后，就可以调用该任务对象的 `add_done_callback` 方法来实现回调函数的绑定。等任务执行完后，系统会自动将任务的结果传入回调函数中的 `future` 参数，通过 `future` 的 `result` 方法即可得到结果

2. 代码实现

代码中，协程处理函数为 `do_some_work`。该函数的功能非常简单：打印当前传入的任务名称，并返回该名称（相当于执行的结果）。回调函数为 `callback`，其功能是将系统传入的结果 `future` 打印出来。代码如下：

代码 10-7：使用协程实现任务提交与结果接收

```
import asyncio
```

```

async def do_some_work(x):                                #定义协程处理函数
    print('任务: ', x)
    return '任务: {}的返回结果'.format(x)

def callback(future):                                     #回调函数
    print('Callback: ', future.result())                 #返回任务结果

coroutine = do_some_work('爬取当天股票')                #定义协程，并传入任务
loop = asyncio.get_event_loop()                         #获得事件循环对象
task = asyncio.ensure_future(coroutine)                 #获得任务对象（对协程的封装）
task.add_done_callback(callback)                        #封装好后的协程对象（任务）就可以绑定回调函数了
loop.run_until_complete(task)                          #执行协程任务

```

将上面代码运行，会输出如下结果：

```

任务： 爬取当天股票
Callback： 任务：爬取当天股票的返回结果

```

第一行的显示，表明接到了处理任务。第二行的显示，表明收到了任务处理后的结果。

10.5 应该选择线程，还是协程

与线程相比，协程封装的内容更多，实现起来也更容易。

开发者使用协程会少考虑很多事情。在默认情况下应优先使用协程。但是有一个特点要注意：

- 协程是多任务的顺序执行，只有当前任务挂起后，才会切换到其他任务来执行。
- 线程是以 CPU 轮训的方式执行。

二者的运行机制有着本质上的区别，这使得二者都不可被代替。即，如果业务需求要求每个并行的线程的处理进度也要同步，那么使用协程将不是一个很好的方案。在实际工作中，应根据业务需要来灵活运用线程或进程。

10.6 实例 38：使用协程批量修改文件扩展名

在 10.4.2 小节中，只是简单地演示了协程的创建。本例在 10.4.2 小节例子的基础上，利用协程批量修改文件扩展名。

实例描述

在本地 D 盘中新建一个文件夹 `test`，并在其中放置若干 Python 代码文件。使用协程批量将这些文件全部转化为扩展名为“`txt`”的文件。

实现协程处理函数 `def change_files`，在其内部使用循环，遍历所有文件，当发现扩展名是“`py`”的文件后，调用 `os` 的 `rename` 方法修其改扩展名。代码如下。

代码 10-8：批量修改文件扩展名

```
import asyncio

async def change_files(x):                                #协程处理函数
    files = os.listdir("D:/test")                        #列出当前目录下所有的文件

    for filename in files:
        portion = os.path.splitext(filename)            #分离文件名字和后缀
        print(portion)

        if portion[1] == ".py":                          #根据后缀来修改，如无后缀则空
            newname = portion[0] + ".txt"                #要改的新后缀
            os.chdir("D:/test")                          #切换文件路径，如无路径则要新建或者路径同上，做好备份
            os.rename(filename, newname)

    return '{}任务完成'.format(x)

def callback(future):                                    #回调函数
    print('Callback: ', future.result())                  #返回任务结果

coroutine = change_files("修改扩展名")                  #定义协程，并传入任务
loop = asyncio.get_event_loop()                          #获得事件循环对象
task = asyncio.ensure_future(coroutine)                  #获得任务对象（对协程的封装）
task.add_done_callback(callback)                          #封装好后的协程对象（任务）就可以绑定回调函数了
loop.run_until_complete(task)                            #执行协程任务
```

代码运行之前，来到 D 盘的 `test` 文件夹下，放置两个 Python 代码文件，如图 10-1 所示。

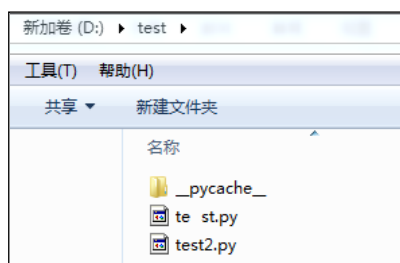


图 10-1 Python 文件

运行程序，显示如下输出：

```
('test', '.py')  
('test2', '.py')  
('__pycache__', '')  
Callback: 修改扩展名任务完成
```

在来到 D 盘的 test 文件夹下，可以看到两个 Python 代码文件的扩展名已经变成“txt”了，如图 10-2 所示。

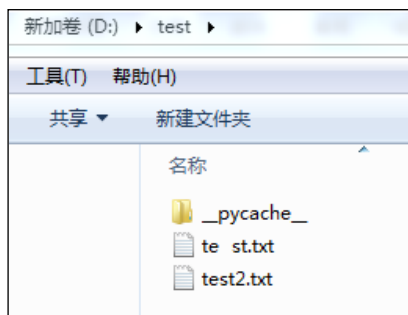


图 10-2 txt 文件

第 4 篇 商业实战

本篇演示了 Python 语言的几个实战案例，帮助读者实现从“技术”到“解决具体问题”的过渡，让读者真真切切地体会到学有所用。

- ▶ 第 11 章 爬虫实战（实例 39）：批量采集股票数据，并保存到 Excel 中
- ▶ 第 12 章 自动化实战（实例 40）：读取 Excel 数据文件，并用可视化分析
- ▶ 第 13 章 机器学习实战（实例 41）：从一组看似混乱的数据中找出 $y \approx 2x$ 的规律
- ▶ 第 14 章 人工智能实战（实例 42）：基于人脸识别的来访登记系统

第 11 章

爬虫实战（实例39）：批量采集股票数据，并保存到Excel中

本章将通过具体的案例，让读者掌握 Python 在爬虫项目中的应用。

通过本章的学习，读者可以掌握分析网页的技巧、Python 编写网络程序的方法、Excel 的操作，以及正则表达式的使用。这些都是爬虫项目中必备的知识 and 技能。

实例描述

通过编写爬虫，将指定日期时段内的全部上市公司股票数据爬取下来，并按照股票代码保存到相应的 Excel 文件中。

这个案例主要分为两大步骤：（1）要知道上市公司有哪些；（2）根据每一个上市公司的股票编号爬取数据。由于两部分代码相对比较独立，可以做成两个代码文件。一个文件用来爬取股票代码，另一个文件用来爬取股票内容。

11.1 爬取股票代码

爬取股票代码的基本思路是：

（1）分析网站上的网页源代码，找到目标代码。

（2）利用正则表达式，在整个网页里搜索目标代码，从而提取出所要的信息（股票代码）。

有关金融证券领域的网站一般都会有上市公司的股票代码信息。随便找一个即可。

11.1.1 找到目标网站

使用 Chrome 浏览器访问链接：<http://quote.eastmoney.com/stocklist.html>。可以看到全部的股票代码，如图 11-1 所示。



图 11-1 股票代码网页

11.1.2 打开调试窗口，查看网页代码

保持当前浏览器窗口为活动页面，按 F12 键显示出网页的源代码调试窗口，单击调试窗口的 Element 按钮，可以看到页面的 HTML 代码，如图 11-2 所示。

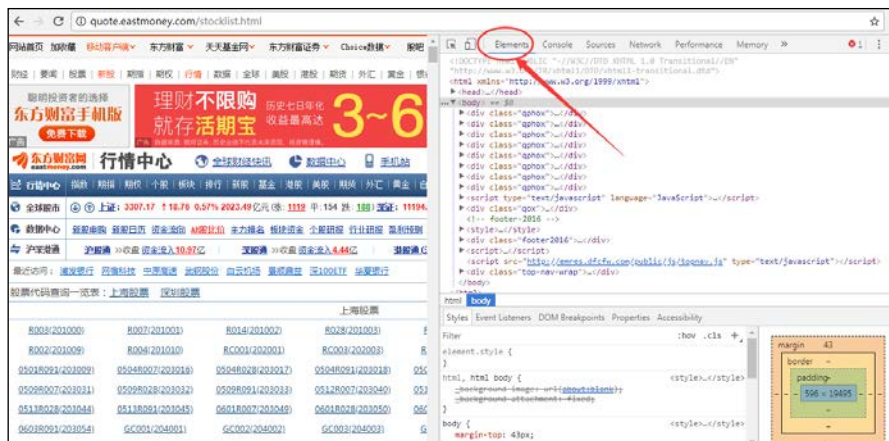


图 11-2 调试窗口

11.1.3 在网页源码中找到目标元素

网页的源代码是按照 HTML 的语法规则自动折叠的。可以用光标在 HTML 代码中任意单击将其展开。当光标移动到某个元素时，会看到右侧网页中对应的元素会有变化，呈现被选中状态，如图 11-3 所示。

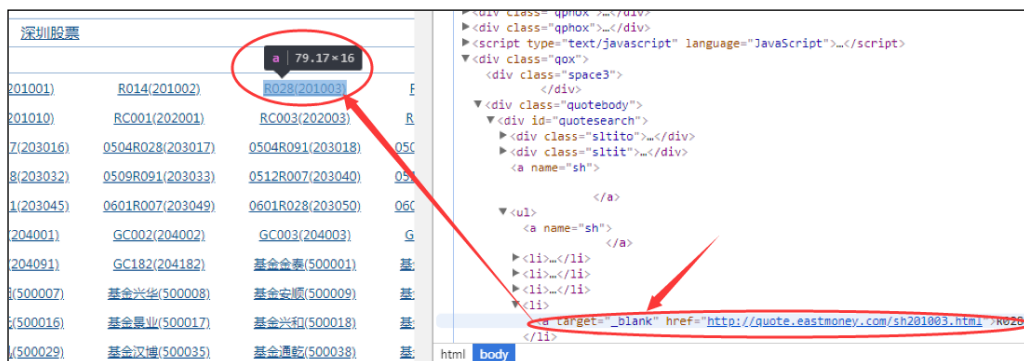


图 11-3 选中元素

图 11-3 中，箭头所指的网页源代码，就是需要关注并爬取的内容。

11.1.4 分析目标源代码，找出规律

分析图 11-3 中左侧显示的内容，与右侧代码之间的对应关系。不难发现，左侧显示的每个股票代码所对应的源代码格式都是固定的，如图 11-4 所示。



图 11-4 目标源代码内容

图 11-4 所示的源代码内容，便是要找到的目标代码。将图 11-4 中的代码整理成如下字符串模板：

```
<li><a target="_blank" href="http://quote.eastmoney.com/股票代码.html
```

其中，“.html”前面的“股票代码”就是需要抓取的内容。分析到这里，开发任务就转化成：在整个网页的源码中，找到这样结构的字符串，并将其中的股票代码提取出来。

11.1.5 编写代码抓取股票代码

编写代码实现 `urlTolist` 函数，并在该函数里实现主要的抓取功能：

- (1) 通过使用 `urllib.request` 模块中的 `urlopen` 函数访问目的链接；
- (2) 通过 `urlopen` 返回值的 `read` 方法获得网页的全部内容；
- (3) 使用 `re` 模块下的 `compile` 函数来做正则表达式的计算模板，其模板字符串就是之前分析的网页目标代码；
- (4) 调用 `re.compile` 返回对象的 `findall` 方法，来对网页的 HTML 代码进行正则表达式计算。得到的返回值 `code` 便是最终的爬取结果。

代码 11-1：爬取股票代码

```

01 import urllib.request                                #网络请求模块
02 import re                                            #正则表达式模块
03
04 stock_CodeUrl = 'http://quote.eastmoney.com/stocklist.html' #要爬取的目的地址
05 def urlTolist(url):                                  #获取股票代码列表
06     allCodeList = []
07     html = urllib.request.urlopen(url).read()         #请求链接，获取网页
08     html = html.decode('gbk')                         #转码
09     s = r'<li><a target="_blank" href="http://quote.eastmoney.com/\S\S(?:.*?)\.html">'
10     pat = re.compile(s)                               #创建正则表达式模板
11     code = pat.findall(html)                          #正则表达式计算
12     for item in code:
13         if item[0]=='6' or item[0]=='3' or item[0]=='0':
14             allCodeList.append(item)
15     return allCodeList                                #返回结果
16
17
18 if __name__=='__main__':
19     allCodelist = urlTolist(stock_CodeUrl)            #调用函数
20     print(allCodelist[:10])                          #显示前 10 条数据

```

在上面代码中，函数 `urlTolist` 的最后 4 行代码是为了让结果更加有效而做的数值验证。即，只有 6（上海证交所）、0（深圳证交所）、3（创业板）打头的股票代码是有效代码。

在代码中，正则表达式的模板部分（代码中的第 9 行），有这么一段代码“`\S\S(?:?)`”。其中，两个“`\S`”表明每个股票代码的前两个都是字符，是要跳过的地方。后面的括号及里面的

内容，表示需要让正则表达式来抓取的部分。（关于正则表达式，不在本书介绍的内容之内，有兴趣的读者可以自行研究。不了解也没关系，直接这么使用即可。）

11.1.6 运行代码，显示结果

代码运行后，显示如下内容

```
['600000', '600001', '600002', '600003', '600004', '600005', '600006', '600007',
'600008', '600009']
```

可以看到，爬取的结果是以 list 的方式存放的。前 10 条都是上海证交所的股票。

11.2 爬取股票内容

通过访问网易提供的服务接口，可以获取到股票内容。只需按照其提供的请求格式，传入股票代码及所要查看的时间段，即可得到该股票的具体数据。

为了爬取全部数据，需要遍历所有的股票代码，并调用网易的服务接口。

11.2.1 编写代码抓取批量内容

在代码实现上，仍然使用 `urllib.request` 模块进行网络请求，并将调用 `urllib.request` 模块下的 `urlretrieve` 函数，将返回的数据保存到 Excel 表里。代码如下：

代码 11-2：爬取股票内容

```
import urllib.request                                #网络请求模块

getstocklist = __import__("11-1 爬取股票代码")      #导入自定义模块
urlTolist = getstocklist.urlTolist

stock_CodeUrl = 'http://quote.eastmoney.com/stocklist.html' #爬取股票代码的目的地址

start = '20161131'                                    #设置查询股票的时间段
end='20161231'

allCodelist = urlTolist(stock_CodeUrl)               #获得全部股票代码
for code in allCodelist:                             #遍历全部代码，调用 163 接口获得数据
    print('正在获取%s 股票数据...' % code)
    if code[0]=='6':
        url = 'http://quotes.money.163.com/service/chddata.html?code=0'+code+\
```

```
'&start='+start+'&end='+end+'&fields=TCLOSE;HIGH;LOW;TOPEN;LCLOSE;CHG;PCHG;TURNOVER
;VOTURNOVER;VATURNOVER;TCAP;MCAP'
else:
    url = 'http://quotes.money.163.com/service/chddata.html?code=1'+code+\'

'&start='+start+'&end='+end+'&fields=TCLOSE;HIGH;LOW;TOPEN;LCLOSE;CHG;PCHG;TURNOVER
;VOTURNOVER;VATURNOVER;TCAP;MCAP'
    urllib.request.urlretrieve(url,'d:\\all_stock_data\\'+code+'_'+end+'.csv')#保存到
Excel
```

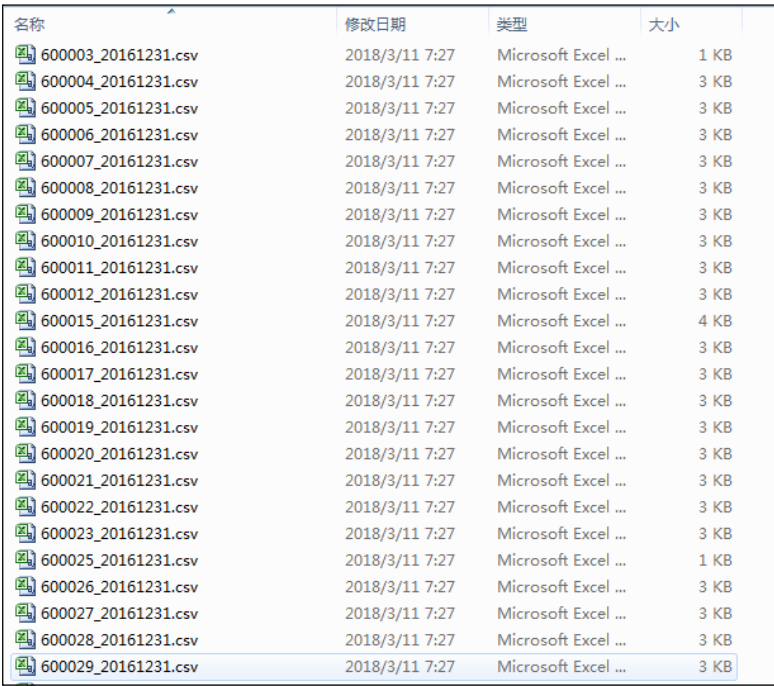
代码中，设置的时间段为 1 个月，即从 20161131 到 20161231。保存的结果放在 D 盘的 all_stock_data 路径下。为了保证保存操作正常运行，需要确保 D 盘下存在 all_stock_data 文件夹（若没有，需要创建一个）。

11.2.2 运行代码显示结果

运行代码，输出如下结果：

```
正在获取 600000 股票数据...
正在获取 600001 股票数据...
正在获取 600002 股票数据...
正在获取 600003 股票数据...
正在获取 600004 股票数据...
正在获取 600005 股票数据...
正在获取 600006 股票数据...
正在获取 600007 股票数据...
正在获取 600008 股票数据...
正在获取 600009 股票数据...
正在获取 600010 股票数据...
正在获取 600011 股票数据...
.....
```

代码运行结束之后，可以在 D 盘的 all_stock_data 文件夹下找到生成的股票数据文件，如图 11-5 所示。



| 名称 | 修改日期 | 类型 | 大小 |
|---------------------|----------------|---------------------|------|
| 600003_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 1 KB |
| 600004_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600005_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600006_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600007_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600008_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600009_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600010_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600011_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600012_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600015_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 4 KB |
| 600016_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600017_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600018_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600019_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600020_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600021_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600022_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600023_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600025_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 1 KB |
| 600026_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600027_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600028_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |
| 600029_20161231.csv | 2018/3/11 7:27 | Microsoft Excel ... | 3 KB |

图 11-5 股票数据文件

至此，爬取股票数据的案例就结束了。读者可以根据上面的知识，在网上爬取更多自己需要的内容。另外，爬取的结果还可以保存到 MySQL 或其他类型文件中。

11.3 爬虫项目的其他技术

爬虫是 Python 语言应用很广的项目方向，涉及到的知识也非常多。因篇幅有限，这里只是举了一个最基本的例子。还有，多线程方式的并发爬取、动态网页的爬取、跨域处理、子链查找、自动登录提权爬取，以及反爬虫处理等多种技术。

在掌握了基本的 Python 知识之后，后面的学习已经不再是难事。有兴趣的读者也可以在该领域深挖细耕，有望成为一个爬虫开发专家，拥有美好的前程。

第 12 章

自动化实战（实例40）：读取Excel数据文件，并用可视化分析

Python 的另一大优势就是自动化处理。它可以使你在处理较为复杂的工作时，节省大量的时间。

实例描述

编写 Python 程序，读取第 11 章所爬取的 Excel 数据文件，并将其按以下要求显示为图，以方便分析。

- （1）将全部数据显示为股市大盘走势图。
- （2）以直方图显示每天的收盘价。
- （3）以折线图显示每天的收盘价与开盘价。

本案例需要使用两个第三方库——Pandas 与 Matplotlib。这两个库都是数据自动化处理中最常用的库。其中，Pandas 专门用于数据的读取、分列、检索等各种复杂操作；Matplotlib 则是用于数据可视化的相关操作。

12.1 使用 Pandas 读取 Excel 文件，并用 Matplotlib 生成大盘走势图

在引入 Pandas 后，直接调用其 `read_csv` 方法，并传入 Excel 文件的指定路径，即可得到 Pandas 类型的数据对象。该对象属于 Pandas 中的 `dataframe` 类型。使用 `dataframe` 类型对象的 `head` 方法可以将其内容显示出来。使用其 `plot` 方法直接可以绘制出对应的大盘走势图。代码如下：

代码 12-1：读取 Excel 数据文件

```
01 from sklearn.manifold import TSNE
02 import matplotlib as mpl
03 import matplotlib.pyplot as plt
04 mpl.rcParams['font.family'] = 'STSong'           #让图示功能支持中文显示
05
06 import pandas as pd                             #导入 Pandas 库
07 df = pd.read_csv("d://all_stock_data//600001.csv",encoding = "gbk") #打开 Excel
08 print(df.head(1))                               #输出部分信息。参数 1，代表显示第 1 条信息
09 df.plot()                                       #输出图片
```

上面的代码中，理论上是可以不引入 Matplotlib 库的，因为 Pandas 中的 dataframe 类型对象有 plot 方法，会将 Matplotlib 集成进来。但是，集成的 Matplotlib 是不支持中文的。需要额外编写代码（见代码的前 4 行），使得绘制的视图能够显示中文。

运行代码，输出如下信息及图片：

| 日期 | 股票代码 | 名称 | 收盘价 | 最高价 | 最低价 | 开盘价 | 前收盘 | 涨跌额 | 涨跌幅 | 换手率 | 成交量 |
|------|------------|---------|--------------|-----|--------------|-----|-----|------|-----|-----|-----|
| 0 | 2009-12-29 | '600001 | 邯郸钢铁 | 0.0 | 0.0 | 0.0 | 0.0 | 5.29 | 0.0 | 0.0 | 0.0 |
| 成交金额 | | | 总市值 | | 流通市值 | | | | | | |
| 0 | 0.0 | | 1.489906e+10 | | 1.489906e+10 | | | | | | |

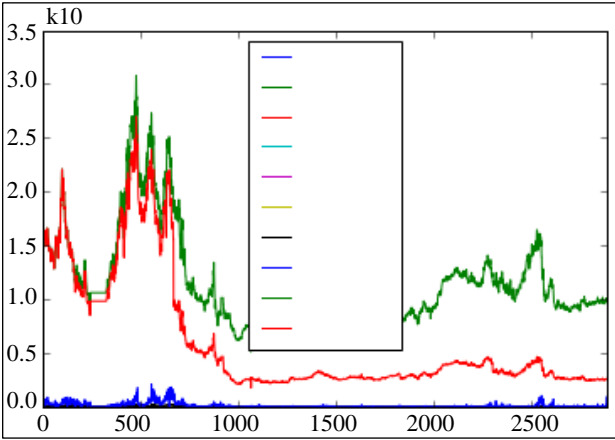


图 12-1 股市大盘走势图

结果中的前两行输出，是由 Pandas 中的 dataframe 类型对象 df 的 head 方法输出的（见代码第 8 行）。在代码第 8 行中，head 的参数是 1，表示将 df 中的第 1 行数据取出。当然，也可以传入整数，以获得对应行数的数据。

图 12-1 显示的内容为，股票数据可视化后的图片。其中，每一种颜色的曲线都是由股票数据中具体的列生成的。纵坐标为所有数值的大小，横坐标为股票数据中条目的顺序索引。

12.2 使用 Pandas 处理数据并绘制成图

下面将演示使用 Pandas 来处理数据，并将数据以直方图和折线图的形式绘制出来。

12.2.1 使用 Pandas 处理数据

数据框（dataframe）类型是 Pandas 中的主要数据类型。对数据中的列进行的操作，基本上都是使用 dataframe 类型对象进行的。例如下列几种情况。

- 选取指定的列：将所要取的列名放到列表里，再将列表放到 dataframe 类型对象后面的中括号里，就会返回另一个只含有指定列的 dataframe 类型对象。
- 修改列名称：将新的列名称放到列表里，并赋值给 dataframe 类型对象的 columns 成员变量。
- 改变列索引：调用 dataframe 类型对象的 set_index 方法，并将指定列的名称传入。

接代码 12-1，编写如下代码：

代码 12-1：读取 Excel 数据文件（续）

| | |
|--|---------|
| 10 price = df[['日期','收盘价']] | #选取关注列 |
| 11 print(price[:5]) | #输出部分信息 |
| 12 price.columns = ["data",'price'] | #修改列名 |
| 13 print(price[:5]) | #输出部分信息 |
| 14 df_new = df[['日期','开盘价','收盘价']].set_index('日期') | #修改索引 |
| 15 print(df_new[:5]) | #输出部分信息 |

上面代码中，可以分为 3 步来看：

（1）从原有的数据框变量 df 中，提取日期列与收盘价列的数据赋值给 price，并将数据框类型的变量 price 中的前 5 条打印出来（见代码 10、11 行）：

| | 日期 | 收盘价 |
|---|------------|-----|
| 0 | 2009-12-29 | 0.0 |
| 1 | 2009-12-28 | 0.0 |
| 2 | 2009-12-25 | 0.0 |
| 3 | 2009-12-24 | 0.0 |
| 4 | 2009-12-23 | 0.0 |

上面的结果显示了数据框变量 `price` 中有 3 列数据：第 1 列是索引，由系统自动编号；第 2 列是日期；第 3 列是收盘价。

(2) 将日期与收盘价的列名分别修改为 `data` 与 `price`，并将前 5 条打印出来。（见代码 12、13 行）输出如下信息：

```

      data price
0  2009-12-29  0.0
1  2009-12-28  0.0
2  2009-12-25  0.0
3  2009-12-24  0.0
4  2009-12-23  0.0

```

结果同样显示了 3 列数据，但是第 2 列与第 3 列的列名变成了 `data` 与 `price`。

(3) 建立一个新的 `dataframe` 类型对象 `df_new`。提取 `df` 的日期列、开盘价列与收盘价列，并将索引设置成日期赋值给变量 `df_new`（见代码 14、15 行）。同样，变量 `df_new` 的前 5 行信息输出如下：

```

      开盘价 收盘价
日期
2009-12-29  0.0  0.0
2009-12-28  0.0  0.0
2009-12-25  0.0  0.0
2009-12-24  0.0  0.0
2009-12-23  0.0  0.0

```

结果显示了 3 列数据：第 1 列为索引，因为不是系统自动生成的，所以该列的名字不是空，而是日期；第 2 列与第 3 列的名字会比索引列的名字高一行，这是为了突出索引列与其他数据列的区别。

12.2.2 绘制直方图和折线图

直方图的绘制非常简单，在 `dataframe` 类型对象的 `plot` 方法中传入一个参数 `kind = 'bar'` 即可。另外，`df` 还可以使用切片来选择指定的数据。见下面代码：

代码 12-1：读取 Excel 数据文件（续）

```

16 df_new['收盘价'][:20].plot(kind = 'bar')      #抽取一列数据的前 20 条，以直方图形式显示
17 df_new[:20].plot()                            #将整个数据的前 20 条，以折线图形式显示

```

上面代码会显示两张图：第 1 幅图是将 `df_new` 的收盘价列数据前 20 条，以直方图形式的

显示（见 16 行）；第 2 幅图是将整个 `df_new` 数据的前 20 条以默认的折线图显示（见 17 行）。

运行代码，得到图 12-2 和图 12-3。

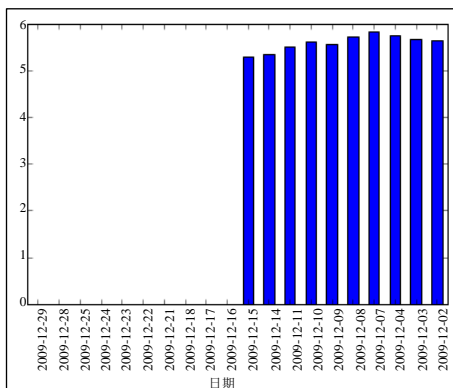


图 12-2 直方图

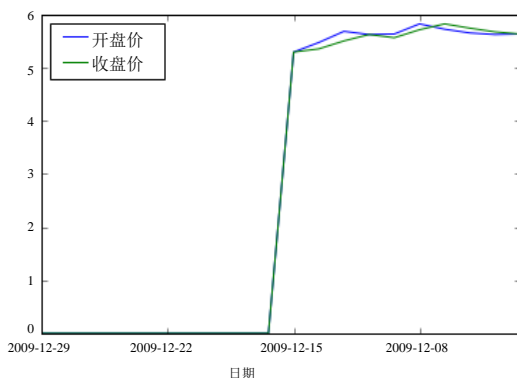


图 12-3 折线图

图 12-2 为直方图的显示。因为 `df_new` 的索引为日期，所以横坐标为具体的日期。而在图 12-3 中存在两个数据。`plot` 函数在绘制时，会自动用不同颜色将其区分开，并会指示不同颜色所指代的数据。

12.3 自动化数据处理的其他技术

本章例子中涉及到的两个工具库，都是目前数据自动化处理领域主流的工具库。

- **Pandas 库**：是个非常强大的数据处理库。这里只是通过实战案例简单地介绍了一下。在 **Pandas** 中，除了 **Dataframe** 类型，还有另一个常用的数据类型——**series**。**series** 类型是负责处理单列级别上的数据；而 **Dataframe** 类型是负责处理列级别的数据。在实际工作中，一般需要将二者结合起来，以实现更高效地自动化数据加工。
- **Matplotlib 库**：是一个功能强大的绘图库。通过简单的几行代码可以绘制点、线、面、二维、三维甚至曲面等各种图像。它是数学科学计算研究过程中必备的可视化工具库。

随着大数据及人工智能学科的兴起，数据的自动化处理的需求也愈发强烈。几乎在各个 IT 行业各个细分领域都可以见到自动化处理。

信息化时代，一定是数据为王。无论是挖掘还是模型训练，都必须在有效的、规整的数据上进行。数据的自动化加工，是“炼数成金”的必经之路。学完 **Python** 基础后，感兴趣的读者也可以在这个方向上继续探索、钻研。

第 13 章

机器学习实战（实例41）：从一组看似混乱的数据中找出 $y \approx 2x$ 的规律

人工智能学科已经成为当今科技的主流。Python 语言之所以能被成为人工智能第一语言，原因在于，其内部封装了大量调用简单、功能强大的库。使得开发人员只需几行代码，就可以完成功能强大的程序，大大地缩短开发时间，提升工作效率。

实例描述

假设有这么一组数据集，看上去很混乱，但 x 和 y 存在着固定的关系。

本例就是让机器学习算法来学习这些样本，并能够找到其中的规律，即，让机器自己能够总结出 $y \approx 2x$ 这样的公式。

本案例将演示机器学习的完整流程。大概分为如下四步：

- （1）准备数据；
- （2）训练模型并实现可视化；
- （3）评估模型；
- （4）保存模型，并应用模型。

13.1 准备数据

这里使用 $y=2x$ 这个公式作为主体，通过加入一些干扰噪声让它的“等号”变成“约等于”。

具体代码思路如下：

- （1）导入头文件，然后生成-1~1 之间的 100 个数作为 x ，见代码第 1~5 行；

(2) 将 x 乘以 2，再加上一个 $[-1,1]$ 区间的随机数乘以 0.3 的积。即， $y = 2 \times x + a \times 0.3$ (a 属于 $[-1,1]$ 之间的随机数)，见代码第 6 行。

代码 13-1：线性回归模型

```
01 import numpy as np
02 import matplotlib.pyplot as plt
03
04 #样本准备
05 train_X = np.linspace(-1, 1, 100)
06 train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 # y=2x, 但是加入了噪声
07 #显示模拟数据点
08 plt.plot(train_X, train_Y, 'ro', label='Original data')
09 plt.legend()
10 plt.show()
```

上面代码中，引入了一个新的第三方模块 **numpy**（见代码第 1 行）。该模块是 Python 在人工智能领域开发的必备模块，是对 Python 中处理数值类型数据的一个升级。使用 **numpy** 处理矩阵相关的线性代数非常方便，能够大大地提升开发效率。



注意：

`np.random.randn(*train_X.shape)` 这段代码如果看起来比较奇怪，现在给出解释——它等同于 `np.random.randn(100)`。这句话的意思是，随机取出 100 个符合标准正态分布特征的数，将其分别与 0.3 相乘。得到的结果将会充当噪声数据，用来干扰整体样本所形成的规律。目的是增加模拟样本的复杂度，使其更接近生活中的自然样本。

现在运行上面代码，显示如图 13-1 所示结果。

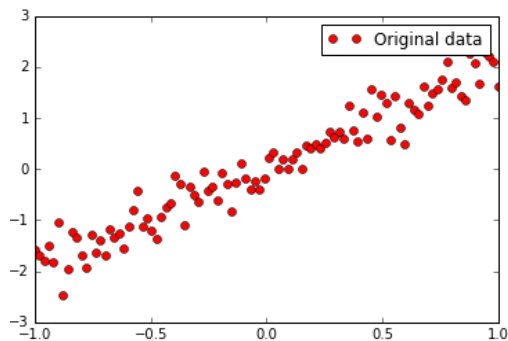


图 13-1 准备好的线性回归数据集

可以看到，样本数据整体趋近于一条直线，但拿出几个具体的点，又不完全符合直线的规律。下面就要用这些数据作为样本，训练模型。

13.2 训练模型并实现可视化

Python 中有一个专门的机器学习算法库——Sklearn。在 Sklearn 里有很多机器学习的算法模型。开发者在使用时，只需要了解模型的应用场景及调用方法，可以完全不理解模型的算法原理。

本例中选取 Sklearn 库中的线性回归模型。线性回归算法，是通过线性模型来对样本进行拟合，将样本回归到某个具体值上。它应用于样本特征趋近与一条直线的场景。

13.2.1 训练模型

有了 Sklearn 库，可以很方便地实现线性回归代码。

首先，将 Sklearn 模块中的线性回归模型 `LinearRegression` 导入，对其进行实例化，生成 `model` 对象。调用 `model` 的 `fit` 方法，传入训练样本。

调用完 `fit` 后，就可以用该模型的 `predict` 方法，对任意输入预算结果了。代码如下：

代码 13-1：线性回归模型（续）

```
11 from sklearn.linear_model import LinearRegression #引入 sklearn 的 LinearRegression
    模型
12 model = LinearRegression()                      #将模型实例化
13 model.fit(train_X.reshape(100,1),train_Y.reshape(100,1)) #进行训练
14 print("输入 6, 的模型预测结果: ",model.predict(6))      #使用模型预测结果
```

第 13 行代码调用了模型对象 `model` 的 `fit` 方法进行模型的训练。其中，传入的参数 `train_X` 与 `train_Y`，都调用了 `reshape` 方法。`reshape` 的意思是，转换数组的形状，即，将原来 1 行 100 列的数组转换成 100 行 1 列的数组。这么做的原因是，为了与模型对象 `model` 的 `fit` 方法相匹配。

13.2.2 利用模型进行预测

在模型训练完之后，传入数值 6，进行模型预测计算（见代码 14 行）。运行程序，将会输出模型预测的结果：

```
输入 6, 的模型预测结果: [[ 12.0851518]]
```

结果显示，模型预测的值为 12.0851518，这说明模型已经基本找到 $y \approx 2x$ 的规律。

13.2.3 了解线性回归模型的内部原理

为了让读者对机器学习不感到陌生，有必要在这里介绍一下模型的内部。

所谓的线性模型，其实就是几何学中的一条直线，见公式：

$$y = kx + b$$

在前面代码中，调用 `model` 的 `fit` 时，系统会根据样本的特征，训练出合适的斜率 k 与截距 b 。当将 6 输入模型进行计算时，公式 13-1 中的 x 等于 6，而 k 与 b 是模型中训练出来的常数，运算之后，最终得到 y 的值。

在代码中，斜率与截距为模型对象 `model` 的两个属性：`coef_`（斜率）与 `intercept_`（截距）。通过代码可以将其值打印出来。下面就编写代码，输出 `coef_` 与 `intercept_` 的值，并将 6 代入直线公式来手动算出预测结果：

代码 13-1：线性回归模型（续）

```
15 print("线性模型的斜率与截距: ",model.coef_,model.intercept_)
16 print("使用斜率与截距的计算结果: ",model.coef_*6 +model.intercept_ ) #y = kx+b
```

代码运行后，输入如下结果：

```
线性模型的斜率与截距: [[ 2.0075617]] [ 0.03978157]
使用斜率与截距的计算结果: [[ 12.0851518]]
```

最后一行的输出为手动算出来的模型结果，与使用模型的 `predict` 方法所生成的结果完全一样。至此，可以更深刻地理解一下机器学习的几个术语。

- 模型：具体的计算公式；
- 训练过程：通过循环迭代不断地修正公式中的参数；
- 预测：将 x 的数值代入，进行计算所得出的结果。

13.2.4 将模型可视化

调用 `Matplotlib` 模块中的方法，将线性回归模型以坐标图的方式显示出来。代码如下：

代码 13-1：线性回归模型（续）

```
17 plt.plot(train_X, train_Y, 'ro', label='Original data')      #显示样本
18 plt.plot(train_X, model.predict(train_X.reshape(100,1)), label='Fitted line')#画
   直线
19 plt.legend()                                                  #显示标注
20 plt.show()                                                    #生成坐标图
```

代码的 17 行，是将样本数据以点的形式显示出来。接着将样本中的 x 与模型运算结果 y 以线的方式显示出来（代码 18 行）。代码运行后，产生图 13-2 所示结果。

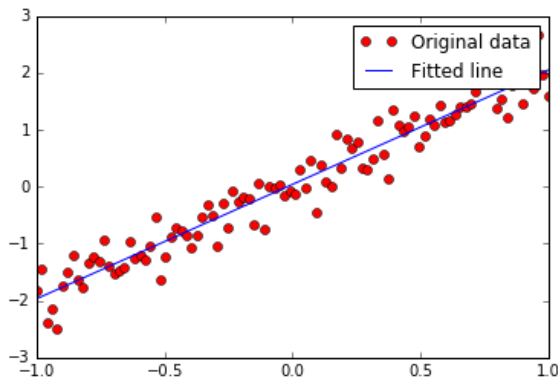


图 13-2 可视化结果

从图 13-2 中可以看出，一条直线几乎处于所有样本点的中心。表明这是相对靠谱的模型。然而，如何知道这个模型是好还是坏呢？在 Sklearn 模块中，还提供了模型评估的方法，见 13.3 节内容。

13.3 评估模型

在机器学习中，模型评估是很必要的的一个手段。只有要经过这个环节，才可以确定该模型是否可用。

13.3.1 评估模型的基本思路

一般来讲，评估模型的基本思路如下：

- (1) 找到一部分测试样本，这部分样本的特征应与训练样本的特征一致；
- (2) 使用模型对测试样本进行计算。并与样本中的真实值比较，评估其正确度。

13.3.2 用代码实现模型评估

在 Sklearn 模块中，全使用模型对象的 score 方法可实现模型的评估。下面通过代码实现：

代码 13-1：线性回归模型（续）

```
21 X_test = np.linspace(11,20,20) #生成测试样本
22 Y_test = 2 * X_test + np.random.randn(*X_test.shape) * 0.3
23 print("模型评估的分值: ",model.score(X_test.reshape(20,1),Y_test.reshape(20,1)))#评估
```

上面代码运行后，输出如下内容：

```
模型评估的分值: 0.997437114846
```

结果显示，模型的准确率达到 99.7%。这属于相当高的分值。这是一个可用的模型。

13.4 保存模型，应用模型

模型训练好了之后，便进入应用阶段。需要先对其进行持久化操作，将其从内存里导出保存起来，放到生产环境下，通过简单的代码载入，并驱使其工作。下面通过代码演示：

代码 13-1：线性回归模型（续）

```
24 from sklearn.externals import joblib
25 joblib.dump(model, "train_model.m") #保存模型
26 model = joblib.load("train_model.m") #载入模型
27 print("导入模型，并输入 6 得到的预测结果: ",model.predict(6)) #使用模型
```

保存模型与保存文件几乎一样，有多种的方法。这里使用的是 Sklearn 库中的 externals 模块。在 externals 下调用 joblib 类的 dump 方法，即可将模型保存到文件中（见代码 25 行）。同样，调用 joblib 类的 load 方法也可以载入模型。使用载入后的模型进行预测运算时，直接调用其 predict 即可。

代码运行后，输出如下内容：

```
导入模型，并输入 6 得到的预测结果: [[ 12.0851518]]
```

该结果与之前内存中的模型预测结果完全一样，表明保存和载入都是正确的。同时，在代码本地的路径下，也会发现一个名为 train_model.m 的文件。这个便是模型文件。

13.5 机器学习的方向

使用 Python 开发机器学习项目时，有两个库是必须要精通的。一个是 Numpy 库，一个是 Sklearn 库。在本章的例子中，只使用了 Sklearn 库中的一个模型。在 Sklearn 中，还有更多的优秀模型，它们在当今的人工智能领域中都有着广泛的使用。

表 13-1 为读者列出了几个 Sklearn 中的常用模型，以方便读者进行研究。

表 13-1 Sklearn 中的常用模型

| 常用模型 | 对应算法的中文名称 |
|------------------------|-----------|
| LogisticRegression | 逻辑回归 |
| GaussianNB | 朴素贝叶斯 |
| KNeighborsClassifier | K 最近邻 |
| svm | 支持向量机 |
| DecisionTreeClassifier | 决策树 |
| RandomForestClassifier | 随机森林 |

机器学习是人工智能的基础。随着科技的发展，在机器学习的基础上，又衍生出更多细分的领域。例如深度学习、强化学习，这都是当前人工智能领域非常热门的领域。

深度学习为人工智能领的技术带来了革命性的飞跃，也是当今最热最火的技术之一。深度学习使用的是深层神经网络的基础结构。对这方面有兴趣的读者，强烈推荐去阅读作者的另一本书《深度学习之 TensorFlow：入门、原理与进阶实战》，其中介绍了包含数值分析、语音、语义、图像处理等多个领域的人工智能知识，总计 96 个案例。该书内容浅显易懂，几乎屏蔽了所有高深的数学理论及生僻术语，可以让零基础读者快速从入门到精通。

第 14 章

人工智能实战（实例42）：基于人脸识别的“来访登记系统”

随着科技的进步，越来越多的办公自动化系统都会和人工智能结合起来。下面就来介绍一个基于人脸识别技术的“来访登记系统”案例。

实例描述

实现一套来访登记系统，要求：能够记录并汇总来访人的到访时间，并且以邮件的方式定期发送给管理员。

对于一个单位来讲，来访人员可以分为两类：体制内的、体制外的。对于体制内的人员，要求能够识别出来具体的人名。对于体制外的人员，要求能够将其照片自动保存并以邮件的形式定期发送给管理员。

本案例属于一个综合性的应用，涉及摄像头调用、图形图像处理、人脸识别模型、多线程、定时器、SMTP 邮件发送协议等多种技术。使用 Python 语言开发这类项目时，无需将每一种技术都实现一遍，而是通过引入相关技术的模块，再配合少量的代码即可实现。下面先从模块的安装开始介绍。

14.1 安装案例所依赖的模块

本案例具体所依赖的模块如下：

- **dlib**：一个强大的机器学习的 C++ 库，包含了许多机器学习常用的算法，同时支持大量的数值算法如矩阵、大整数、随机数运算等。在本案例中，它属于间接依赖，为 `face_recognition` 模块提供支撑。
- **face_recognition**：一个人脸识别库。该模块使用 `dlib` 模块中最先进的人脸识别深度学习

算法，使得识别准确率在《Labeled Faces in the world》测试基准下达到了 99.38%。在本案例中，它属于直接依赖，提供人脸识别功能。

- `face_recognition_models`: `face_recognition` 模块所使用的模型文件。
- `opencv`: 计算机视觉模块，封装了大量的视觉处理算法，安装轻量且运行高效。在本案例中，它属于直接依赖，提供调用摄像头采集数据，并显示图片功能。
- `yagmail`: 实现发邮件功能的模块。使用它，可以非常简单地实现自动发邮件功能。在本案例中，它属于直接依赖，负责向管理员发送来访人员的信息。

以上几种模块都需要独立安装。除此之外，本案例还依赖 `time`、`datetime`、`os`、`numpy`、`PIL`、`threading` 等模块，这些模块在 `Anaconda` 中已经集成，不需要额外安装。

由于 `opencv` 在显示图片时不支持中文字体，还需要额外下载中文字体并进行集成。本案例中会以黑体字体 `simhei.ttf` 为例。

下面依次介绍各个模块的安装。

14.2 安装及使用 `dlib` 模块

`dlib` 模块的安装步骤与一般模块的安装步骤不同。虽然也可以使用 `pip` 命令进行安装，但往往都失败。这是因为，`dlib` 模块的官方网站并没有给出可以直接安装的最新版本。即使利用 `pip` 命令进行安装，也需要将其源码下载，并在本地编译。而在编译 `dlib` 的过程中，系统又需要引入大量的第三方软件包，常常会因为本地缺少或没有合适的第三方软件包，而发生安装错误。

使用离线方式下载具有安装包版本的 `dlib` 模块，会是一个比较省劲的选择。具体可以分为两个步骤：

- (1) 下载 `dlib` 离线安装包。
- (2) 使用 `pip` 命令进行离线安装。

14.2.1 下载 `dlib` 模块

直接下载安装包并进行安装，会比源码编译更节省时间，且不容易出错。在 `pypi` 的官网上，可以找到支持 Python 各个版本的 `dlib` 模块安装包。但是它们都不是最新版本。`Dlib` 的 18.17.1 版本提供了支持 Python 2.7、3.4、3.5 版本的投资包。`dlib` 的 19.7.0 版本提供了支持 Python 3.6 版本的安装包。如果想使用最新的 `dlib` 模块，也可以自行下载源代码进行编译。下面是关于各

个版本的安装包及最新版本源码的下载方法。

1. 下载 Dlib 的 18.17.1 版本

dlib 的 18.17.1 版本的链接地址：<https://pypi.python.org/pypi/dlib/18.17.100>。

访问上述链接，可以找到有关 dlib 库 18.17.100 版本在各个 Python 版本下的安装包，它们包括了 Windows 32 位系统及 64 位系统，如图 14-1 所示。

| File | Type | Py Version | Uploaded on | Size |
|--|--------------|------------|-------------|------|
| dlib-18.17.100-cp27-none-win32.whl (md5) | Python Wheel | cp27 | 2015-09-27 | 1MB |
| dlib-18.17.100-cp27-none-win_amd64.whl (md5) | Python Wheel | cp27 | 2015-09-29 | 1MB |
| dlib-18.17.100-cp34-none-win32.whl (md5) | Python Wheel | cp34 | 2015-09-29 | 1MB |
| dlib-18.17.100-cp34-none-win_amd64.whl (md5) | Python Wheel | cp34 | 2015-09-29 | 1MB |
| dlib-18.17.100-cp35-none-win32.whl (md5) | Python Wheel | 3.5 | 2016-01-29 | 1MB |
| dlib-18.17.100-cp35-none-win_amd64.whl (md5) | Python Wheel | 3.5 | 2016-01-29 | 1MB |
| dlib-18.17.100.zip (md5) | Source | | 2015-09-29 | 4MB |

图 14-1 dlib 的 18.17.1 版本

2. 下载 Dlib 的 19.7.0 版本

dlib 的 19.7.0 版本的链接地址：<https://pypi.python.org/pypi/dlib/19.7.0>。

访问上述链接，可以找到 dlib 库 19.7.0 版本在 Python 3.6、Windows 64 位操作系统环境下的安装包，如图 14-2 所示。

| File | Type | Py Version | Uploaded on | Size |
|--|--------------|------------|-------------|------|
| dlib-19.7.0-cp36-cp36m-win_amd64.whl (md5) | Python Wheel | cp36 | 2017-09-17 | 2MB |
| dlib-19.7.0.tar.gz (md5) | Source | | 2017-09-17 | 3MB |

图 14-2 dlib 的 19.7.0 版本

从图 14-2 中可以看到，dlib 库 19.7.0 版本只有一个在 Python 3.6、Windows 64 位操作系统环境下的安装包。直接将 dlib-19.7.0-cp36-cp36m-win_amd64.whl 下载即可。为了配合后面所使用的 face_recognition 模块，本案例选择下载 dlib 库的 19.7.0 版本，并使用 Python 3.6 环境在 Windows 下实现。

3. 下载 Dlib 最新版本的源码

如要使用 dlib 的最新版本，只能下载源码进行编译了。可以通过如下链接访问官网：<http://dlib.net>。

在浏览器中打开该网站后，可以找到有关 dlib 的最新版本下载链接以及编译方法，如图 14-3 所示。

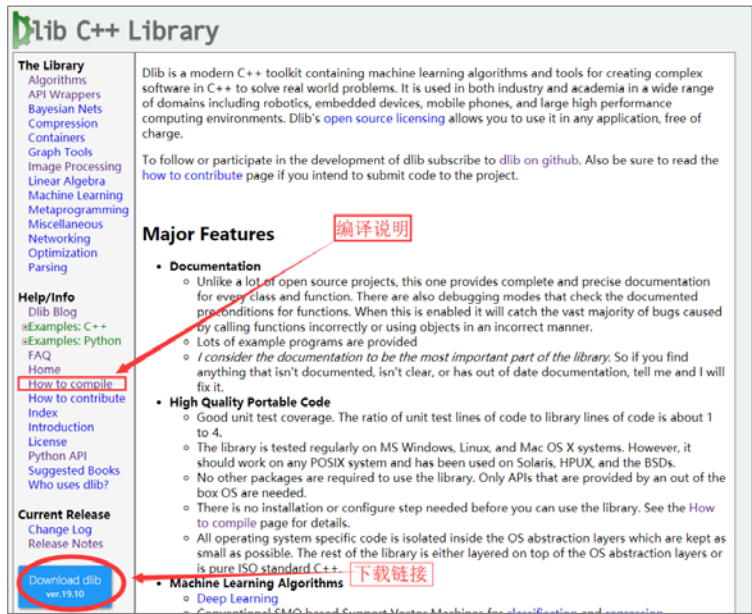


图 14-3 dlib 的最新版本

由于 dlib 是用 C++语言编写，编译时需要下载对应的 C++编译器。具体环节可以自行研究。

本例中使用的是 dlib 库 19.7.0 版本的安装包，如果想要与本案例同步操作，也可以下载 dlib 库 19.7.0 版本。

14.2.2 安装 dlib 模块

下载完安装包 dlib-19.7.0-cp36-cp36m-win_amd64.whl 后，来到安装包所在的目录下（本例中的路径为 C:\Users\ljh\Downloads），执行如下命令即可安装成功：

```
pip install dlib-19.7.0-cp36-cp36m-win_amd64.whl
```

输入该命令后，系统会提示安装成功，如图 14-4 所示。

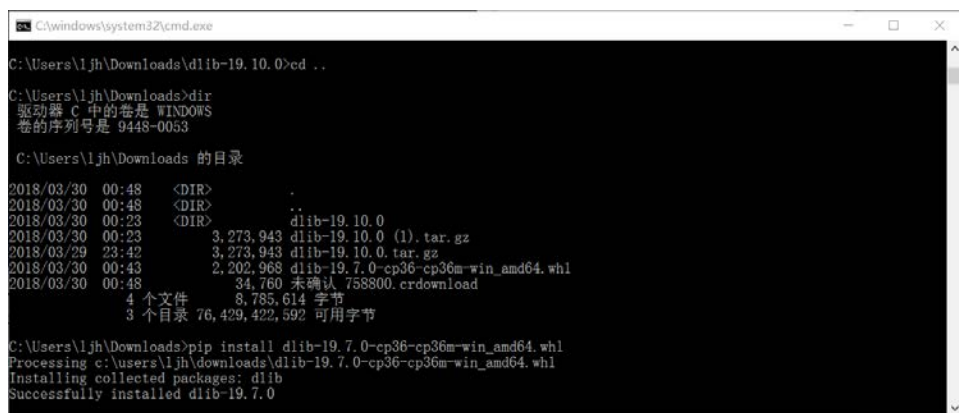


图 14-4 dlib 安装成功

图 14-4 所示，是在 Windows 10 环境操作系统上、Python 3.6 版本环境下，使用 pip 命令成功安装 dlib 模块的截图。



注意：

不建议在 Windows 下使用 pip 进行安装。因为，如果使用 pip install dlib 命令进行安装，系统会默认地从官网下载源代码，然后在本机编译。由于 dlib 的源码中自动编译的兼容性做得不是很好（尤其是 Windows 下），常常会因为缺少库，或依赖的库版本不一致导致编译失败。

14.2.3 使用 dlib 模块进行人脸检测

安装好 dlib 后，就可以通过简单的几行代码来实现人脸检测功能。如下：

```

import dlib
from skimage import io

detector = dlib.get_frontal_face_detector() #使用 dlib 自带的 frontal_face_detector
作为我们的特征提取器

win = dlib.image_window() #使用 dlib 提供的图片窗口
img = io.imread("2.jpg") #使用 skimage 的 io 读取图片
dets = detector(img, 1) #使用 detector 进行人脸检测 dets 为返回的结果
win.set_image(img) #显示图片(dlib 的 ui 库可以直接绘制 dets)
win.add_overlay(dets) #在显示的图片上绘制人脸标注框

```

将一个人物图片（2.jpg）放到代码的同级目录下，运行代码后，所生成的结果如图 14-5 所示。



图 14-5 dlib 人脸检测

从图 14-5 所示的结果中可以看出，dlib 库还是很强大的。即使人脸是倾斜的，也可以很容易地检测出来。

14.3 安装及使用 face_recognition 模块

安装 face_recognition 模块，可以直接使用 pip 命令。

14.3.1 下载 face_recognition 模块

在使用 pip 安装 face_recognition 的过程中，会关联安装其他的依赖模块（例如 face_recognition_models），整个执行过程如图 14-6 所示。

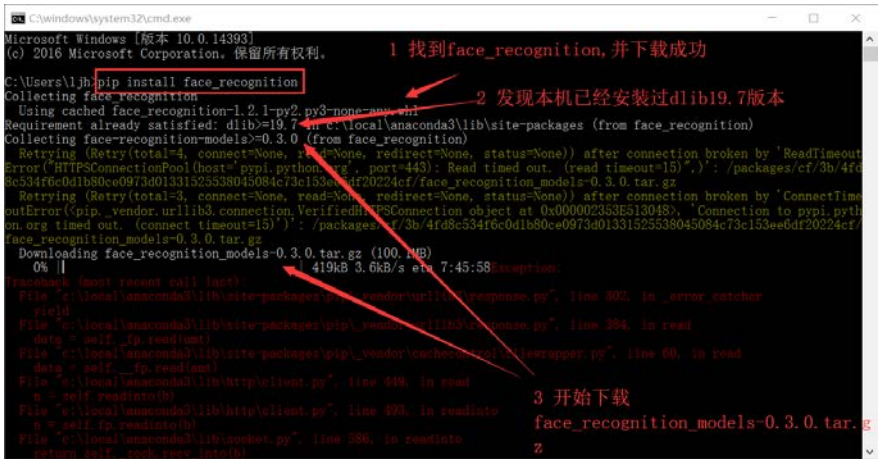


图 14-6 face_recognition 模块安装过程

在图 14-6 中可以看出，使用 pip 命令进行安装 face_recognition 模块时，系统进行了 3 步操作：

(1) 找到 face_recognition 并下载。

(2) 安装 dlib 库 19.7.0 版本（这步操作一般都会失败，建议提前按照前面 14.2 节内容单独安装）。

(3) 下载并安装 face_recognition_models-0.3.0.tar.gz 模型（由于网速环境等因素，这步操作也很容易出错，建议手动安装。见 14.3.2 小节）。

这里，在执行安装 face_recognition 的第（3）步时，已经出错了（见图 14-6 中标注 3）。于是采取手动安装的方式，补齐 face_recognition_models-0.3.0.tar.gz 模型。

14.3.2 下载及安装 face_recognition_models 模型

访问网站：https://pypi.python.org/pypi/face_recognition_models。

找到 face_recognition_models-0.3.0.tar.gz 模型安装包的下载地址，如图 14-7 所示。



图 14-7 face_recognition_models 模型下载页面

单击图 14-7 中箭头所指的绿色按钮，下载 face_recognition_models-0.3.0.tar.gz 模型安装包。

安装包下载完之后，进行解压，然后编译、安装，如图 14-8 所示。

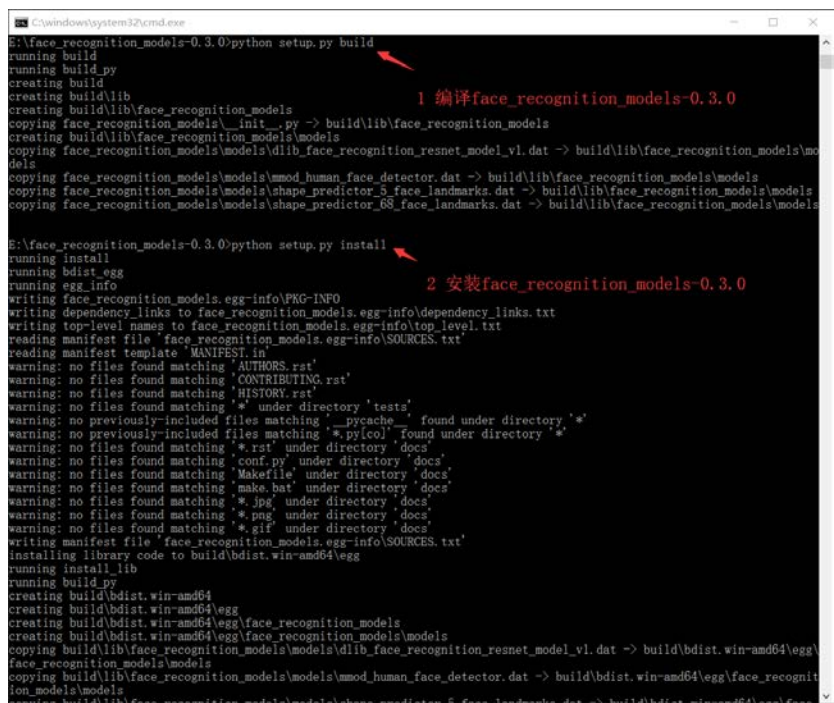


图 14-8 face_recognition_models 模型安装页面

本例中，face_recognition_models-0.3.0.tar.gz 模型安装包的解压路径为 E:\face_recognition_models-0.3.0。按照图 14-8 中箭头所标示的两个步骤进行安装：

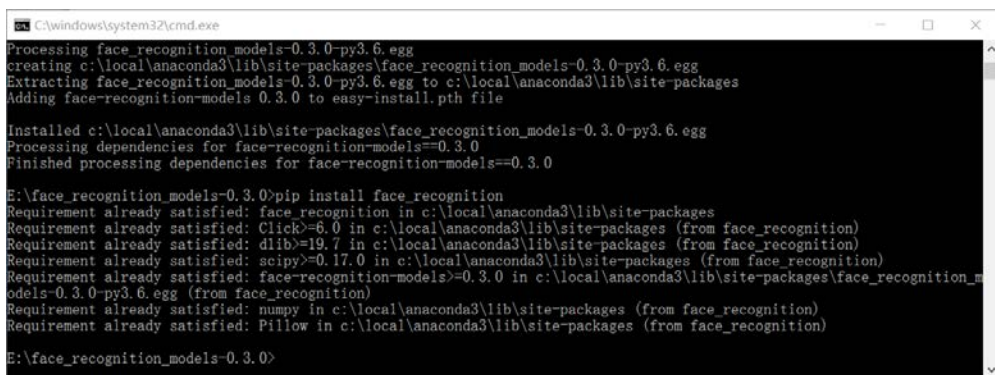
- (1) 执行命令：python setup.py build，进行编译。
- (2) 执行命令：python setup.py install，进行按装。

安装成功后，会有输出提示，如图 14-9 所示。



图 14-9 face_recognition_models 模型安装成功

face_recognition_models-0.3.0.tar.gz 模型安装成功之后，理论上 face_recognition 包就安装结束了。为了确认 face_recognition 是否正确安装，再执行一下 face_recognition 的安装命令，如图 14-10 所示。



```

C:\windows\system32\cmd.exe
Processing face_recognition_models-0.3.0-py3.6.egg
creating c:\local\anaconda3\lib\site-packages\face_recognition_models-0.3.0-py3.6.egg
Extracting face_recognition_models-0.3.0-py3.6.egg to c:\local\anaconda3\lib\site-packages
Adding face_recognition_models-0.3.0 to easy-install.pth file

Installed c:\local\anaconda3\lib\site-packages\face_recognition_models-0.3.0-py3.6.egg
Processing dependencies for face_recognition_models==0.3.0
Finished processing dependencies for face_recognition_models==0.3.0

E:\face_recognition_models-0.3.0>pip install face_recognition
Requirement already satisfied: face_recognition in c:\local\anaconda3\lib\site-packages
Requirement already satisfied: Click>=6.0 in c:\local\anaconda3\lib\site-packages (from face_recognition)
Requirement already satisfied: dlib>=19.7 in c:\local\anaconda3\lib\site-packages (from face_recognition)
Requirement already satisfied: scipy>=0.17.0 in c:\local\anaconda3\lib\site-packages (from face_recognition)
Requirement already satisfied: face_recognition_models>=0.3.0 in c:\local\anaconda3\lib\site-packages\face_recognition_models-0.3.0-py3.6.egg (from face_recognition)
Requirement already satisfied: numpy in c:\local\anaconda3\lib\site-packages (from face_recognition)
Requirement already satisfied: Pillow in c:\local\anaconda3\lib\site-packages (from face_recognition)

E:\face_recognition_models-0.3.0>

```

图 14-10 face_recognition 模块安装成功

图 14-10 中显示，执行完 pip install face_recognition 之后，没有出现任何报错信息，这表明安装正确。

14.3.3 使用 face_recognition 模块检测人脸中的特征点

下面使用 face_recognition 模块检测人脸特征点，代码如下：

```

from PIL import Image, ImageDraw
import face_recognition  # 导入 face_recognition 模块

image = face_recognition.load_image_file("ss.jpg")  # 将 jpg 文件加载到 numpy 数组中
face_landmarks_list = face_recognition.face_landmarks(image) # 获得面部特征(可能会识别出多个人)

for face_landmarks in face_landmarks_list:          # 描绘出每个人的面部特征
    pil_image = Image.fromarray(image)              # 转为 Image 类型
    d = ImageDraw.Draw(pil_image)                   # 画人物原始图片
    for facial_feature in face_landmarks.keys():     # 遍历人类各个器官的特征位置
        d.line(face_landmarks[facial_feature], width=5) # 画出特征

    pil_image.show()                                # 显示图片

```

本例中，将一个文件名为“ss.jpg”的图片放在代码的统计目录下。上面代码运行后，显示的结果如图 14-11 所示。

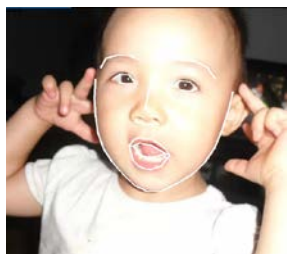


图 14-11 face_recognition 模块检测人脸特征点

提取人脸特征点，是人脸识别的基础。有了特征点之后，不仅可以实现人脸识别，还可以做更多实用的功能了，如开发美颜、换脸等功能。

14.4 安装及使用 opencv 模块

opencv 支持 Python 版本的模块叫作 opencv-python，可直接使用 `pip install opencv-python` 命令进行安装，但常常会因为网络等问题导致失败。推荐使用离线模式来安装 opencv 模块。具体做法同样是：先下载，再安装。

14.4.1 下载并安装 opencv 模块

这里使用的是 opencv 最新的版本 3.4.0.12，在如下网站进行下载：
<https://pypi.python.org/pypi/opencv-python/3.4.0.12>。

下载完安装包后，使用如下命令进行安装：

```
D:\>pip install opencv_python-3.4.0.12-cp36-cp36m-win_amd64.whl
```

因为这个离线安装的过程与前面 dlib、face_recognition 的安装类似，此处不再过多描述。

14.4.2 下载中文字体

在 4.1 节讲到过，opencv 不支持中文的输出，需要下载中文字体自行安装。具体下载链接是：http://www.font5.com.cn/font_download.php?id=151&part=1237887120。

打开网页，单击图 14-12 中箭头所指的链接即可下载。



图 14-12 下载中文字体

下载好的字体文件是个压缩包，将其解压后，会得到一个名为 `simhei.ttf` 的文件。将其复制到代码的同级目录下即可。

14.4.3 使用 opencv 调用摄像头进行拍照

下面通过几行代码，来用 `opencv` 模块调用摄像头进行拍照。具体如下：

```
import cv2                                #导入 cv 模块
from PIL import Image, ImageDraw, ImageFont  #导入 PIL 模块
import numpy as np                        #导入 numpy 模块

cap = cv2.VideoCapture(0)                #打开摄像头
while(1):
    ret, frame = cap.read()                #获得一帧视频图像
    img_PIL=Image.fromarray(cv2.cvtColor(frame,cv2.COLOR_BGR2RGB)) #将图片转为 PIL 支持的格式
    font = ImageFont.truetype('simhei.ttf', 40) #载入字体
    draw = ImageDraw.Draw(img_PIL)         #画出原始图片
    draw.text((100,100), '按q键拍照并退出', font=font, fill= (255, 255, 255)) #在图片上面显示中文
    frame = cv2.cvtColor(np.asarray(img_PIL),cv2.COLOR_RGB2BGR) #将图片转为 cv 支持的格式
    cv2.imshow("capture", frame)            #将这一帧图像显示出来
    if cv2.waitKey(1) & 0xFF == ord('q'):   #捕捉键盘输入，当输入q时保存文件并退出
        cv2.imwrite("out.jpg", frame)
        break
cap.release()                             #释放资源
cv2.destroyAllWindows()                   #关闭窗口
```

将 14.4.2 小节下载的中文字体 `simhei.ttf` 放到代码的同级目录下，并保证本机安装有摄像头。运行该代码，会弹出摄像头采集窗口，按 `q` 键之后，程序退出。在代码的同级目录下会找到一个名为 `out.jpg` 的图片文件，如图 14-13 所示。



图 14-13 opencv 调用摄像头拍照

在图 14-13 上，可以看到有汉字显示。这个字体就是来自 14.4.2 小节所下载的 simhei.ttf 字体文件。

14.5 安装及使用 yagmail 模块

yagmail 模块的功能是发送邮件。其安装方法非常简单，直接使用 pip 即可。yagmail 模块的使用也非常简单，几行代码即可。

14.5.1 安装 yagmail 模块

可以使用在线的方式安装 yagmail 模块。直接一条 pip 命令即可完成 yagmail 安装。如下：

```
pip install yagmail
```

14.5.2 使用 yagmail 模块向自己的 QQ 邮箱发送邮件

使用 yagmail 模块发送邮件的代码相对简单，主要分为三步：

- (1) 设置自己的 QQ 邮箱，使其支持 SMTP 服务。
- (2) 调用代码进行登录。
- (3) 向目的邮箱发送邮件。

第 (1) 步中的 SMTP 是邮件发送协议。在邮箱中打开 SMTP 服务后，邮箱就可以支持用户使用第三方工具发送邮件了。具体如下。

1. 设置自己的 QQ 邮箱，使其支持 SMTP 服务

(1) 通过 Web 方式登录到自己的 QQ 邮箱，单击“设置”按钮，然后单击“账户”按钮，如图 14-14 所示。



图 14-14 QQ 邮箱设置界面

(2) 在图 14-14 界面中找到图 14-15 的选项，单击图中箭头所指的“开启”按钮，开启 POP3/SMTP 服务。QQ 邮箱的设置中是将 POP3 服务与 SMTP 服务一起开启的。该按钮包含了本案例中需要用到 SMTP 服务，



图 14-15 开启 QQ 邮箱的 POP3/SMTP 服务

(3) 弹出窗口，进入验证密保环节。要求使用已关联的手机，向目的号码发送一条短信，短信内容为“配置邮件客户端”，如图 14-16 所示。

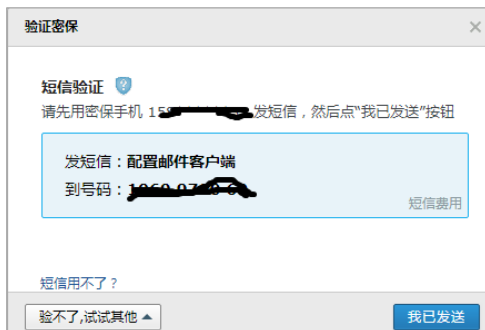


图 14-16 QQ 邮箱验证密保

(4) 按照图 14-16 的要求，使用手机发送短信之后，单击图 14-16 中的右下方“我已发送”按钮。系统接着会弹出成功界面，上面有显示使用第三方 POP3/SMTP 服务登录时的密码，如图 14-17 所示。



图 14-17 QQ 邮箱 SMTP 服务开启成功

图 14-17 所示的虚线框中的字符串，即为使用 SMTP 服务的登录密码。将该字符串复制。使用代码完成剩下的发送功能。

2. 编写代码实现登录

yagmail 中的登录代码只有一行。如下：

```
import yagmail #导入 yagmail 模块
yag = yagmail.SMTP("QQ 号码@qq.com", '登录密码', 'smtp.qq.com', 465) #进行登录
```

上面的代码中，将 yagmail 模块的 SMTP 类进行实例化，得到了对象 yag，同时也完成了登录。实例化的参数有 4 个。

- 第一个：QQ 邮箱。填入自己的 QQ 邮箱即可。
- 第二个：登录密码。见图 14-17 中虚线框中的字符串。
- 第三个：SMTP 服务器。这是固定域名，无需改动。
- 第四个：SMTP 服务器的端口。也无需改动。

3. 发送邮件

发送邮件的代码也是只有一行。具体如下：

```
yag.send(['收件人 qq@qq.com', '收件人 qq @qq.com'], 'test', 'efgh', ['2-1 命令行参数.py', '3-2 调用模块.py'])
```

代码中，yag 为前面“2. 编写代码实现登录”中 yagmail.SMTP 实例化返回的对象。其 send 方法所支持的参数如下。

- 第一个：收件人邮箱，列表类型。
- 第二个：邮件主题，字符串类型。
- 第三个：邮件正文，字符串类型。
- 第四个：附件，列表类型，里面的元素为文件的路径字符串。
- 第五个：抄送人邮箱，列表类型。
- 第六个：抄送人邮箱，列表类型。

其中后 3 个参数为可选参数。

将“2. 编写代码实现登录”与“3. 发送邮件”这两部分代码合起来后，就是一个完整的发送邮件程序。运行后可以在收件人的 QQ 邮箱中找到对应的邮件，如图 14-18 所示。



图 14-18 QQ 邮箱收件界面

图 14-18 所示，该信箱收到了一个标题为“test”、内容为“efgh”，并带有两个附件的邮件。

14.6 详细设计

为了能让多模块间合理地集成起来，在编写软件之前，详细设计的环节必不可少。本案例

的详细设计环节较为简洁，只是以实用出发，并没有严格的套搬软件工程学中的正规流程。

14.6.1 需求描述

见 14 章开头“案例描述”部分。

14.6.2 定义系统的输入和输出

这是个相对较为独立的系统，输入和输出都比较清晰。具体如下。

- 输入：硬件摄像头设备传入的视频流。
- 输出：邮件。格式见表 14-1。

表 14-1 邮件格式

| 邮件元素 | 格式及说明 |
|------|--|
| 标题 | 来访统计记录 |
| 内容 | 以多行的形式表示，每一行代表一条记录。每条记录包括人名和时间，中间用逗号分隔。 具体如下： 人名 1，时间 1 人名 2，时间 2 |
| 附件 | 放置未识别别人的人脸图片 |

14.6.3 系统规则及约束

系统的细节规则制定及约束如下。

- 统计汇报时间：24 小时向管理员发送一次来访记录；
- 单人单次记录的时间间隔：在 10 分钟之内出现某个人的多条记录，视为一次访问记录。

14.6.4 结构体设计

根据表 14-1 的输出内容，设计程序结构体如下。

- 单条记录：定义单条记录类 Recorder，用于实现具体某一条记录的存放。
- 汇总记录：定义字典类型全局变量 red_dict，用于来存放全部的来访信息。其中，key 为具体人名，value 为类 Recorder 的实例对象。

- 未识别图片记录：定义列表类型全局变量 `unknownjpg`，用于来存放未识别图片的文件名。

14.6.5 软件的主体架构图

软件分为主循环体、定时器线程、`face_recognition` 模块、`yagmail` 模块、`facelib` 文件夹五大部分。其结构关系如图 14-19 所示。

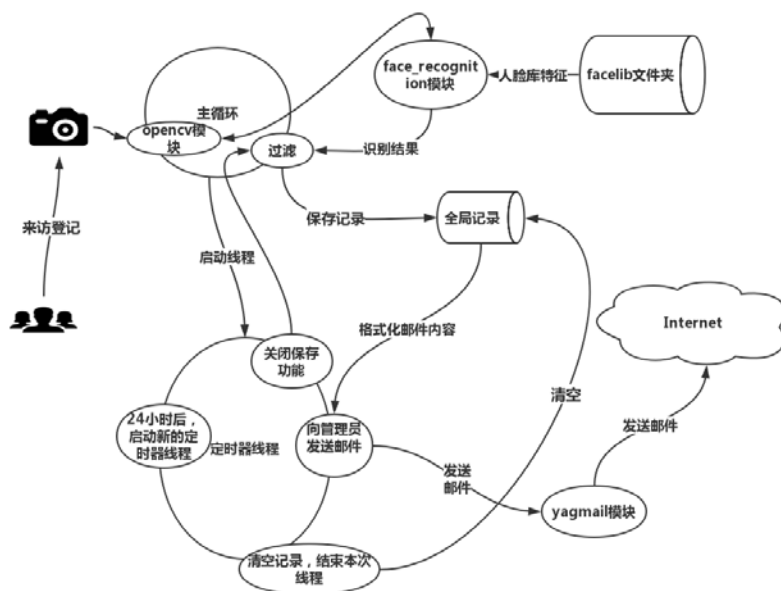


图 14-19 软件架构图

其中的各部分的作用如下。

- 主循环体：负责整个程序的逻辑主控制。
- 定时器线程：负责阶段性数据汇总及报告的相关流程。
- `face_recognition` 模块：负责进行人脸识别。
- `yagmail` 模块：负责发送邮件。
- `facelib` 文件夹：存放体制内人员的照片信息。文件名字需要与照片中的人名相对应。人脸识别模块将根据该文件夹中的人物来识别来访者身份。

14.6.6 软件的主体流程介绍

根据图 14-19 的软件架构图，可将软件的主体流程描述为以下步骤。

- (1) 主程序启动，创建定时器线程，并启动主循环。
- (2) 在主循环里，调用摄像头采集视频数据。
- (3) 在主循环里，将采集到的视频数据传入 `face_recognition` 模块进行人脸识别。
- (4) 在主循环里，对人脸识别返回的结果进行过滤并保存。
- (5) 在定时器线程里，创建一个新的定时器，并在 24 小时之后执行。
- (6) 在定时器线程里，通过修改全局标志，停止主线程的保存记录工作。
- (7) 在定时器线程里，调用邮件模块，将保存的记录发送。
- (8) 在定时器线程里，清空已保存的记录。
- (9) 在定时器线程里，开启主线程的保存记录功能，并退出本次定时器线程。
- (10) 待到 24 小时之后，第 (5) 步创建的新定时器线程又开始启动，接着重复步骤 (5)~(10)。

14.7 编码实现

在实现本案里的技术储备与详细设计后，则进入了编码阶段。

14.7.1 导入模块

将前面所列出的模块全部导入进来，具体如下：

代码 14-1：基于人脸识别的来访登记系统

| | |
|--|--------------------------------|
| 01 import face_recognition | #导入 face_recognition 模块，用于人脸识别 |
| 02 import cv2 | #导入 opencv 模块，用于摄像头调用及显示 |
| 03 import os | |
| 04 import numpy as np | |
| 05 from PIL import Image, ImageDraw, ImageFont | #导入 PIL 模块，用于中文显示 |
| 06 | |
| 07 import datetime | |
| 08 import threading | #导入 threading 模块 |
| 09 import time | |
| 10 import yagmail | #导入 yagmail 模块，用于发送邮件 |

14.7.2 定义结构体

单条记录的结构体类 `Recorder` 被定义为一个空的类。这样做的好处是，便于以后扩展。变量 `red_dict` 与 `unknownjpg` 的值为空，用于存放来访记录和未识别图片的文件名称。

代码 14-1：基于人脸识别的来访登记系统（续）

```

11 class Recorder:                                #定义单条记录的结构体类
12     pass
13
14 red_dict = {}                                    #定义全局变量存放来访记录
15 unknownjpg = []                                #定义全局变量存放未识别文件名

```

14.7.3 实现发送邮件函数

封装函数 `sendemail` 的功能是发送邮件，其中参数有三个，分别为：邮件标题（`title`）、邮件内容（`contents`）、附件列表（`fileslist`）。

代码 14-1：基于人脸识别的来访登记系统（续）

```

16 def sendemail(title,contents,fileslist):
17     yag = yagmail.SMTP("发件人邮箱",'密码','smtp.qq.com', 465)
18     yag.send(['收件人邮箱 1','收件人邮箱 2'],title,contents,fileslist)

```

在编写上面代码时，需要将代码 17 行中的“发件人邮箱”与“密码”换成自己的邮箱和密码。并将“收件人邮箱 1”与“收件人邮箱 2”换成真正要发送的目的邮件地址。

14.7.4 实现邮件内容生成函数

封装函数 `dicttostr` 的功能是，将全局变量 `red_dict` 中的数据生成固定格式的字符串，用作邮件内容来使用，返回值是一个包含具体记录的列表，列表中的每条记录都是一个字符串。

代码 14-1：基于人脸识别的来访登记系统（续）

```

19 def dicttostr():
20     strlist = []
21     listkey =list(sorted(red_dict.keys()))        #取字典的 key
22     for item in listkey:                          #通过循环，合成每一条的来访记录
23         strlist.extend([item+' '+str(onetime) for onetime in red_dict[item].times])
24     return strlist

```

14.7.5 实现过滤并保存来访记录的函数

封装函数 `saveRecorder` 的功能是，过滤并保存来访记录。对于未识别的用户，将其图片保存下来。

代码 14-1：基于人脸识别的来访登记系统（续）

```

25 flagover = 0                                #全局标志，用来控制是否保持来访记录
26 def saveRecorder(name,frame):
27     global red_dict
28     global flagover
29     global unknownjpg
30     if flagover==1:                            #响应全局标志，如果为 1 时，关闭保存记录的功能
31         return
32     try:
33         red = red_dict[name]                    #如果是多次识别，比较时间
34         secondsDiff=(datetime.datetime.now()-red.times[-1]).total_seconds()
35
36         if secondsDiff<60*10:                  #如果两次的时间在 10 分钟以内，将被过滤掉
37             return
38         red.times.append(datetime.datetime.now()) #添加符合规定的到访记录
39         print('更新记录',red_dict,red.times)
40     except (KeyError):                          #如果是当天第一次来访，将走以下分支
41         newRed = Recorder()                     #新建一个记录
42         newRed.times=[datetime.datetime.now()]
43         red_dict[name] =newRed                  #将记录保存
44         print('添加记录',red_dict,newRed.times)
45
46     if name == 'Unknown':                      #如果是未识别
47         s =str(red_dict[name].times[-1])
48         print('写入',s[:10]+s[-6:])
49         filename = s[:10]+s[-6:]+'.jpg'
50         cv2.imwrite(filename,frame)             #将图片保存
51         unknownjpg.append(filename)             #添加到访记录

```

14.7.6 实现定时器处理函数

封装函数 `loop_timer_headle` 功能是，定时器线程的处理函数。该函数主要完成以下几件事：

- (1) 通过设置变量 `flagover` 为 1，来关闭保存记录功能。
- (2) 创建一个新的定时器，令其在 24 小时后启动。
- (3) 调用函数 `sendemail` 将邮件发送出去。

(4) 清空所保存的记录。

(5) 打开保存记录功能。

具体代码如下。

代码 14-1：基于人脸识别的来访登记系统（续）

```

52 def loop_timer_headle():                                #定时器循环触发函数
53     global timer2                                       #载入全局变量
54     global flagover
55     global red_dict
56     global unknownjpg
57
58     flagover = 1                                         #关闭保存记录功能
59     timer2 = threading.Timer(60*60*24, loop_timer_headle) #创建定时器线程， 24 小时之后
    执行
60     timer2.start()                                     #启动定时器线程
61     sendemail("来访统计记录",'\\n'.join(dicttostr()),unknownjpg) #发送邮件
62     print("清空")
63     red_dict.clear()
64     unknownjpg.clear()
65
66     print("重新开始")
67     flagover = 0                                         #打开保存记录功能

```

14.7.7 在主线程中启动定时器线程

使用 `threading` 的 `Timer` 类实例化定时器 `timer2`，并调用 `timer2` 的 `start` 方法启动定时器线程。
代码如下：

代码 14-1：基于人脸识别的来访登记系统（续）

```

68 timer2 = threading.Timer(2, loop_timer_headle) #两秒钟后执行线程处理函数
    loop_timer_headle
69 timer2.start()                                         #启动线程

```

14.7.8 实现并调用函数载入人脸库

定义函数 `load_img`，其功能为将文件夹 `facelib` 中的人脸照片载入，并使用 `face_recognition` 将其特征编码返回，与文件名对应。代码如下：

代码 14-1：基于人脸识别的来访登记系统（续）

```

70 def load_img(sample_dir):
71     for (dirpath, dirnames, filenames) in os.walk(sample_dir):#一级一级地文件夹递归
72         print(dirpath,dirnames,filenames)
73         facelib = []                                     #获取人脸库图片
74         for filename in filenames:
75             filename_path = os.sep.join([dirpath, filename])
76             print(filename_path)
77             faceimage = face_recognition.load_image_file(filename_path)
78             face_encoding = face_recognition.face_encodings(faceimage)[0] #每个图只取
    一人
79             facelib.append(face_encoding)                 #保存人脸库特征
80     return facelib,filenames
81
82 facelib,facename = load_img('facelib')                  #调用函数载入人脸库

```

该代码有个默认的规则，就是文件夹 `facelib` 中放置的照片必须要清晰，并且是人物的正脸。照片的文件名要与该人姓名相同。

在使用 `face_recognition` 模块进行人脸比对时，就会根据 `facelib` 中最相似的图片索引，找到对应的图片名称，从而定位到具体某个人。

14.7.9 在主循环里调用摄像头，并进行人脸识别

最后一个功能是启用一个循环，调用摄像头，将每一帧的图片放到定义函数 `load_img`。其功能为，将文件夹 `facelib` 中的人脸照片载入，并使用 `face_recognition` 将其特征编码返回，与文件名对应。代码如下：

代码 14-1：基于人脸识别的来访登记系统（续）

```

83 video_capture = cv2.VideoCapture(0)    #获得摄像头
84
85 face_locations = []                    #定义列表存放人脸位置
86 face_encodings = []                    #定义列表存放人脸特征编码
87 process_this_frame = True              #定义信号量
88
89 while True:
90     ret, frame = video_capture.read()    #捕获一帧图片
91     small_frame = cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)#将图片缩小为原来的 1/4,
    为人脸识别提速
92     rgb_small_frame = small_frame[:, :, ::-1]    #将 opencv 的 BGR 格式转为 RGB 格式
93

```

到此，整个代码就编写完毕。在代码的同级目录下，还需要放置好字体文件“simhei.ttf”与文件夹“facelib”。文件夹“facelib”中放置的是已知人员的照片。具体如图 14-20 所示。



图 14-20 facelib 中人脸库

在图 14-20 中可以看到，每张照片的文件名都要与人物对应。人脸识别模块最终会把照片的文件名与识别结果关联起来。运行程序之前，还要注意下邮箱的设置，确保程序可以自动发送邮件。

14.8 运行程序

一切准备就绪后，便可以运行程序查看效果了。为了测试方便，这里把各个环节的时间调短了一些：

- 14.7.5 小节中代码第 36 行，将多次访问的时间间隔设为 20 秒。
- 14.7.6 小节中代码第 59 行，将上报来访信息的时间设为 1 分钟。

程序运行后，屏幕就会弹出窗口。当捕捉到人脸时，就会显示该人的名字，如图 4-21 所示。

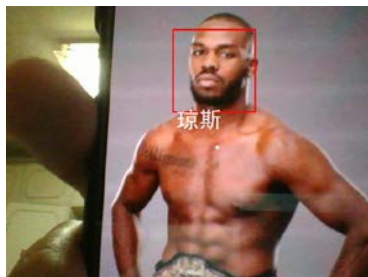


图 14-21 人脸识别结果

图 14-21 所示的人物是著名 UFC 运动员琼斯的照片，人脸识别程序精确地将其识别出来。

在摄像头窗口下，按 q 键可以退出程序。

确保程序运行时间超过在一分钟之后，就可以检查自己的目的邮箱，在里面可以看到系统自动发送的邮件。具体内容如图 14-22 所示。



图 14-22 收到的来访信息邮件

图 14-22 可以看到：邮件的标题为“来访统计记录”；邮件中的正文便是访客的访问记录；附件为当时没有识别的访客图片。

14.9 下一步对系统的改进

该系统只是完成了来访登记的主体基础功能，在很多细节上还存在着较大的改进空间。例如下列四方面。

- 加入活体检测：在 14.8 节可以看到，单独使用一个手机上的照片就可以将来访登记系统骗过。这种系统在真正应用时是有漏洞的。可以在人脸特征点（见 14.3.3 小节）基础之上，加入活体检测算法，以避免使用照片骗过系统的现象。
- 添加更多的生物特征检查：可以加入声纹和指纹识别来配合人脸识别，以实现更精确的身份识别功能。其中，Dejavu 库可以实现声纹识别，而指纹识别也有各种对应的算法，

具体可以参考 `fingerPrint` 的实现。

- 添加硬盘数据的问题功能：现在的系统只是单纯地将未识别的人物照片保存起来，对于一个长期运行的系统来讲，保存的照片越来越多，最终会写满整个磁盘。为了防止这种现象，可以设置图片留存的期限，将过时的图片定期清除。
- 以人物为单位记录未识别人：当前的系统只是以图片为单位保存未识别人。这会造成大量的信息冗余。更好的方式是：在众多未识别人物的图片中，进行一次人脸识别，将零散的照片以人物的方式统计出来。这样会使系统上报的信息更加实用、简洁。

附录 A

内置函数

Python 3.5 版本中共有 68 个内置函数，见表附件-1。

表附件-1 算术运算符

| 分 类 | 数 量 |
|------|-----|
| 数学运算 | 7 |
| 类型转换 | 25 |
| 序列操作 | 7 |
| 对象操作 | 9 |
| 反射操作 | 8 |
| 变量操作 | 2 |
| 交互操作 | 2 |
| 文件操作 | 1 |
| 编译执行 | 4 |

下面按照表附件-1 的内容依次详细展开。

1. 数学运算内置函数

表附件-2 数学运算内置函数

| 函 数 | 描 述 |
|--------------|--|
| abs(x) | 取 x 的绝对值，例如：abs(-2)，结果为 2 |
| max(x,[key]) | 返回可迭代对象中的元素中的最大值或者所有参数的最大值。例如： max(1,2,3)，结果为 3 max(-1,0,key = abs)，绝对值后的最大数，结果为-1 |
| min(x,[key]) | 返回可迭代对象中的元素中的最小值或者所有参数的最小值。例如：min(1,2,3)，结果为 1 min(-1,-2,key = abs)，绝对值后的最小数，结果为-1 |

续表

| 函 数 | 描 述 |
|---------------------------|---|
| <code>pow(x, y)</code> | 幂计算，等价与 <code>**</code> 。 例如： <code>pow(2,3)</code> 等价与 <code>2**3</code> ，即 3 的 4 次方，结果为 8 |
| <code>sum(x)</code> | 对可迭代对象 <code>x</code> 的所有元素求和。例如： <code>sum((1,2,3,4))</code> ，结果为 10 <code>sum((1,2,3,4),-2)</code> ，传入 <code>sum</code> 进行相加的对象可以嵌套，结果为 8 |
| <code>divmod(x, y)</code> | 返回商和余数。例如： <code>divmod(13, 4)</code> ，结果为(3, 1) |
| <code>round(x,n)</code> | 对浮点数 <code>x</code> 进行四舍五入求值,取小数点后 <code>n</code> 位。例如： <code>round(1.1314926,5)</code> ，结果为 1.13149 |

2. 类型转换内置函数

表附件-3 类型转换内置函数

| 函 数 | 描 述 |
|----------------------------------|--|
| <code>int(x,[base])</code> | 将 <code>x</code> 转换为整型。 <code>x</code> 可以是字符串或其他数字； <code>base</code> 是可选参数，默认为 10，表示将字符串 <code>x</code> 转化为 10 进制整数。 当 <code>base</code> 被赋值时， <code>x</code> 必须是字符串；当 <code>x</code> 为浮点数时，转成的整数会将小数点后面全部舍掉。 如果想要更精确的转化，推荐用 <code>math</code> 库里面的 <code>floor</code> 和 <code>ceil</code> 函数来明确转换方式 |
| <code>float(x)</code> | 将 <code>x</code> 转换为浮点型 |
| <code>bool(x)</code> | 根据传入的参数的逻辑值创建一个新的布尔值。例如： <code>bool(0)</code> ，结果为 <code>False</code> ； <code>bool(1)</code> ，结果为 <code>True</code> |
| <code>str(x)</code> | 返回一个对象的字符串表现形式。例如： <code>str(123)</code> ，结果为'123' |
| <code>bytearray(x,encode)</code> | 根据传入的参数创建一个新的字节数组， <code>encode</code> 代表编码。 例如： <code>bytearray('中文','utf-8')</code> ，结果为 <code>b'\xe4\x88\xad\xe6\x96\x87'</code> |
| <code>bytes(x,encode)</code> | 根据传入的参数创建一个新的不可变字节数组， <code>encode</code> 代表编码。例如： <code>bytes('中文','utf-8')</code> ，结果为 <code>b'\xe4\x88\xad\xe6\x96\x87'</code> |
| <code>memoryview(x)</code> | 根据传入的参数创建一个新的内存查看对象。 例如： <code>v = memoryview(b'abcefg')</code> <code>Print(v[1])</code> ，结果为 98 |
| <code>ord(x)</code> | 返回 <code>Unicode</code> 字符对应的整数。例如： <code>ord('a')</code> ，结果为 97 |
| <code>chr(x)</code> | 返回整数所对应的 <code>Unicode</code> 字符。例如： <code>chr(97)</code> ，结果为'a' |
| <code>bin(x)</code> | 将整数转换成二进制字符串。例如： <code>bin(3)</code> ，结果为'0b11' |
| <code>oct(x)</code> | 将整数转换成八进制字符串。例如： <code>oct(10)</code> ，结果为'0o12' |

续表

| 函 数 | 描 述 |
|---|---|
| <code>hex(x)</code> | 将整数转换成十六进制字符串。例如： <code>hex(15)</code> ，结果为 <code>'0xf'</code> |
| <code>complex(re,im)</code> | 生成复数。 <code>re</code> 为实数部分， <code>im</code> 为虚数部分。例如： <code>complex(8,7)</code> ，则生成一个复数 <code>8+7j</code> |
| <code>range(start, stop[, step])</code> | 根据传入的参数创建一个新的 <code>range</code> 对象，从 <code>start</code> 开始到 <code>stop</code> 结束。 <code>step</code> 为步长，即每隔 <code>step</code> 个数取一次。 例如： <code>list(range(6))</code> ，结果为 <code>[0, 1, 2, 3, 4, 5]</code> <code>list(range(1,6))</code> ，结果为 <code>[1, 2, 3, 4, 5]</code> <code>list(range(1,10,2))</code> ，1 到 10 之间每隔两个位取一次数，结果为 <code>[1, 3, 5, 7, 9]</code> |
| <code>tuple(x)</code> | 根据传入的参数创建一个新的元组。 例如： <code>tuple()</code> ，不传入参数，表示创建空元组 <code>tuple('123')</code> ，结果为 <code>('1', '2', '1')</code> |
| <code>list(x)</code> | 根据传入的参数创建一个新的列表。 例如： <code>list()</code> ，不传入参数，表示创建空列表 <code>list('123')</code> ，结果为 <code>['1', '2', '1']</code> |
| <code>set(x)</code> | 根据传入的参数创建一个新的集合。 例如： <code>set()</code> ，不传入参数，表示创建空集合 <code>set(range(3))</code> ，结果为 <code>{0, 1, 2 }</code> |
| <code>frozenset(x)</code> | 根据传入的参数创建一个新的集合。 例如： <code>frozenset(range(3))</code> ，结果为 <code>frozenset({0, 1, 2 })</code> |
| <code>dict(x)</code> | 根据传入的参数创建一个新的字典。 例如： <code>dict()</code> ，不传入参数，表示创建空字典 <code>dict(a = 1,b = 2)</code> ，结果为 <code>{'b': 2, 'a': 1 }</code> <code>dict(zip(['a','b'],[1,2]))</code> ，结果为 <code>{'b': 2, 'a': 1 }</code> <code>dict(((('a',1),('b',2))))</code> ，结果为 <code>{'b': 2, 'a': 1 }</code> |
| <code>enumerate(x, start)</code> | 根据可迭代对象创建枚举对象。例如： <code>list(enumerate([22,33,44]))</code> ，结果为 <code>[(0, 22), (1, 33), (2, 44)]</code> <code>list(enumerate([22,33,44], start=1))</code> ，结果为 <code>[(1, 22), (2, 33), (3, 44)]</code> |
| <code>iter(x)</code> | 根据传入的参数创建一个新的可迭代对象。之后可以使用 <code>next</code> 函数来取值。当取值结束后，再次调用 <code>next</code> 会触发 <code>StopIteration</code> 异常。 例如： <code>a = iter('1234')</code> ， <code>next(a)</code> ，结果为： <code>'1'</code> |
| <code>next(x)</code> | 与 <code>iter</code> 配合使用。见 <code>iter</code> 例子 |
| <code>slice(start, stop[, step])</code> | 根据传入的参数创建一个新的切片对象。例如： <code>print(slice(1,10,3))</code> ，结果为： <code>slice(1, 10, 3)</code> |
| <code>super()</code> | 根据传入的参数创建一个新的子类和父类关系的代理对象。通过对象可以调用父类的方法。例如： <code>super().__init__()</code> ，表示调用父类的初始化函数 |

3. 序列操作内置函数

表附件-3 序列操作内置函数

| 函 数 | 描 述 |
|---|---|
| all(iterable) | 判断可迭代对象的每个元素是否都为 True 值，如果输入的 iterable 内为空，也会返回 True。例如： all(), 空元组，返回 True; all([9,2]), 列表中每个元素逻辑值均为 True，返回 True; all([0,1,2]), 列表中 0 的逻辑值为 False，返回 False |
| any(x) | 判断可迭代对象的元素是否有为 True 值的元素，如果输入的 iterable 内为空，会返回 False。例如： any(), 空元组，返回 False; any([0,0]), 列表中每个元素逻辑值均为 False，返回 False; any([0,2]), 列表中 1 的逻辑值为 True，返回 True |
| map(x) | 使用指定方法去作用传入的每个可迭代对象的元素，生成新的可迭代对象。见本书 6.3.4 小节。 |
| filter(x) | 使用指定方法过滤可迭代对象的元素。见本书 6.3.4 小节。 |
| reversed(x) | 反转序列生成新的可迭代对象。例如： print(list(reversed([1,2,3])))，输出[3, 2, 1] |
| sorted(iterable, key=None, reverse=False) | 对可迭代对象进行排序，返回一个新的列表。其中，参数 key 可以指定比较函数，reverse 可以指定排序的顺序。例如：print(list(sorted([1,4,2])))，输出[1, 2, 4] |
| zip (x) | 聚合传入的每个迭代器中相同位置的元素，返回一个新的元组类型迭代器。例如： x = [1,2] y = [4,5,7] list(zip(x,y))，结果为[(1, 4), (2, 5)] |

4. 对象操作内置函数

表附件-4 对象操作内置函数

| 函 数 | 描 述 |
|-----------|---|
| help(str) | 返回对象的帮助信息。例如：help(sorted)，显示 sorted 函数的帮助文档 |
| dir(x) | 返回对象或者当前作用域内的属性列表。例如：dir(math)，返回 math 模块中的符号 |
| id(x) | 返回对象的指针（唯一标识符）。见本书 4.1.1 与 4.2.8 小节。 |

续表

| 函 数 | 描 述 |
|---|--|
| <code>hash(x)</code> | 获取对象的哈希值。例如： <code>hash('Python')</code> ，得到一个 3166564652540415167 的数字。 |
| <code>type(x)</code> | 返回对象的类型，或者根据传入的参数创建一个新的类型。见本书 4.1.1 与 4.2.1 小节。 |
| <code>len(x)</code> | 返回对象的长度。例如： <code>print(len('Python'))</code> ，输出 6 |
| <code>ascii(x)</code> | 返回对象的可打印表字符串表现方式。例如： <code>ascii(1)</code> ，结果为 '1' |
| <code>format(value, format_spec=", /")</code> | <p>格式化显示值。<code>format_spec</code> 表示所要转换的形式，取值如下：'s'代表字符串； 'b'代表二进制； 'c'代表 unicode 字符串； 'd'代表十进制； 'o'代表八进制； 'x'代表 16 进制，并将转化后的字母小写； 'X'代表 16 进制，并将转化后的字母大写； 'n'代表十进制； 'e'代表科学计数法，并将转化后的字母小写； 'E'代表科学计数法，并将转化后的字母大写； 'f'小数点计数法，并将转化后的字母小写； 'F'小数点计数法，并将转化后的字母大写； 'g'代表先用科学计数法格式化，得到的幂指数在-4 与原来数值之间，就改用小数计数法，并将转化后的字母小写； 'G'代表先用科学计数法格式化，得到的幂指数在-4 与原来数值之间，就改用小数计数法，并将转化后的字母大写。（科学技术法：把原来的数值变为一个小数与一个 10 的 n 次幂的形式表示。小数计数法：保留原数值对小数点后一定位数。）</p> <p>例如： <code>format(3, 'b')</code>，结果为 '11'</p> |
| <code>vars([object])</code> | 返回当前作用域内的局部变量和其值组成的字典，或者返回对象的属性列表。例如： <code>vars(str)</code> ，返回 <code>str</code> 类的属性列表 |

5. 反射操作内置函数

表附件-5 反射操作内置函数

| 函 数 | 描 述 |
|---|---|
| <code>__import__(str)</code> | 动态导入模块。见本书 3.5.4 小节 |
| <code>isinstance(x)</code> | 判断对象是否是类或者类型元组中任意类元素的实例。见本书 9.5.1 小节 |
| <code>issubclass(cls, class_or_tuple, /)</code> | 判断类是否是另外一个类或者类型元组中任意类元素的子类。见本书 9.5.2 小节 |
| <code>hasattr(x)</code> | 检查对象是否含有属性。见本书 9.5.3 小节 |
| <code>getattr(x)</code> | 获取对象的属性值，见本书 9.5.4 小节 |
| <code>setattr(x)</code> | 设置对象的属性值。见本书 9.5.5 小节 |
| <code>delattr(x)</code> | 删除对象的属性 |
| <code>Callable(x)</code> | 检测对象是否可被调用 |

6. 变量操作内置函数

表附件-6 变量操作内置函数

| 函 数 | 描 述 |
|------------------------|---|
| <code>globals()</code> | 返回当前作用域内的全局变量和其值组成的字典。例如： <code>print(globals())</code> ，返回全局变量 |
| <code>locals()</code> | 返回当前作用域内的局部变量和其值组成的字典。例如： 在某个函数里 <code>print(locals())</code> ，返回当前函数的局部变量 |

7. 交互操作内置函数

表附件-7 交互操作内置函数

| 函 数 | 描 述 |
|---|-----------------------|
| <code>print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)</code> | 向标准输出对象打印输出。见本书 4.3 节 |
| <code>input([prompt])</code> | 读取用户输入值。见本书 4.3 节 |

8. 文件操作内置函数

表附件-8 文件操作内置函数

| 函 数 | 描 述 |
|--|---------------------------------------|
| <code>open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)</code> | 使用指定的模式和编码打开文件，返回文件读写对象。 见本书 8.1 节 |

9. 编译执行内置函数

表附件-9 编译执行内置函数

| 函 数 | 描 述 |
|--|---|
| <code>compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)</code> | 将字符串编译为代码或者 AST 对象，使之能够通过 <code>exec</code> 语句来执行或者 <code>eval</code> 进行求值。例如： <code>code1 = 'i = 5; print (i)'</code> <code>compile1 = compile(code1, '', 'exec')</code> <code>exec (compile1)</code> ，输出 5 |
| <code>eval(source, globals=None, locals=None, /)</code> | 执行动态表达式求值。见本书 6.6 节 |
| <code>exec(source, globals=None, locals=None, /)</code> | 执行动态语句块。见本书 6.6 节 |
| <code>repr(obj, /)</code> | 返回一个对象的字符串表现形式（给解释器）。见本书 9.6.1 小节 |

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34322>

